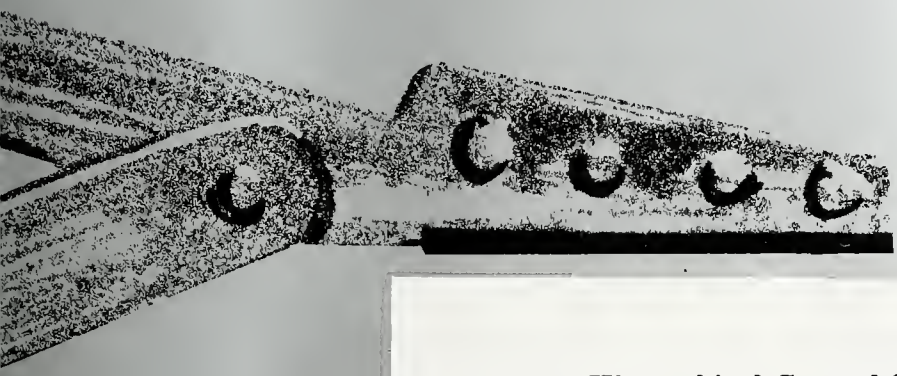
Robotics Research Technical Report

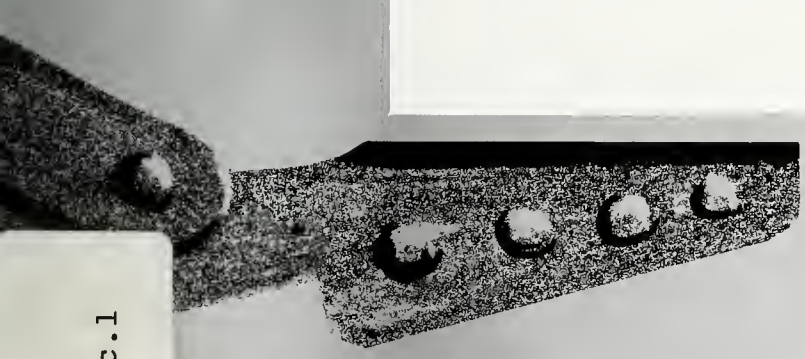


Hierarchical Control System

by

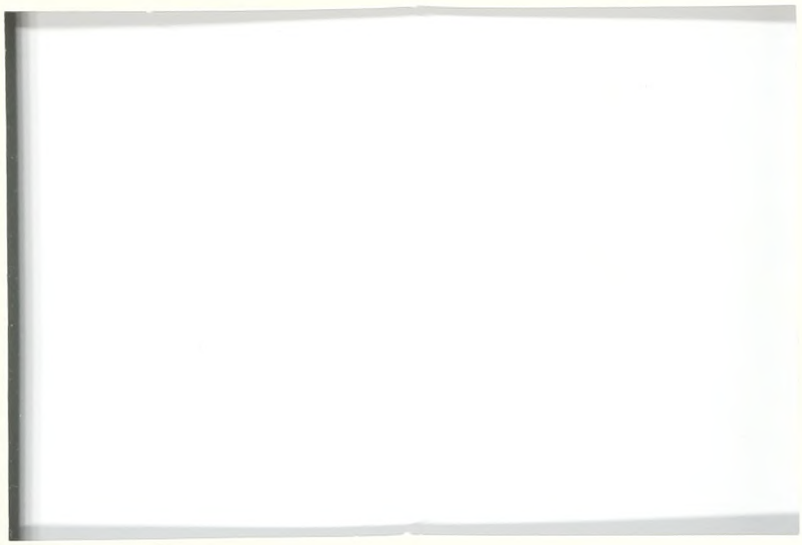
Dayton Clark

Technical Report No. 396
Robotics Report No. 167
February, 1989



New York University
rant Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012



Hierarchical Control System

by

Dayton Clark

Technical Report No. 396
Robotics Report No. 167
February, 1989

New York University
Dept. of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, New York 10012

Work on this paper has been supported by Office of Naval Research Grant N00014-87-K-0129 National Science Foundation CER Grant DCR-83-20085, National Science Foundation Grant subcontract CMU-406349-55586, and by grants from the Digital Equipment Corporation and the IBM Corporation.

Abstract

Servo loops are widely used in robot control programs. The Hierarchical Control System (HIC) is an operating system for managing a hierarchy of servo loops. HIC is in use at NYU in the controller for the Utah/MIT hand. Key features of HIC are a simple and efficient scheduler for servos and asynchronous events and a (nominally) non-blocking message passing structure, the periodic data buffer. This report describes HIC in detail with examples of its use.

Contents

Section 1

Introduction

The Hierarchical Control System (HIC) is a real-time operating system for robot control. In particular HIC is intended for those parts of the systems performing the lower level control functions, joint servoing, sensor input, the aggregation of joints and sensors into meaningful units (fingers, arms, etc.) and so on. Programs at this level of control frequently have the following characteristics: the processes or tasks are typically not asynchronous but are scheduled at regular intervals; program flow is of the nature, *input* — *compute* — *output*; the computations are of what might be called "closed form" which here means that essentially the same calculation is performed each cycle, it takes about the same amount of time each cycle, there are no references to complex or remote data bases, there are no indefinite searches of large problem spaces, and there is very little if any interaction with a user or operator; and there are often "hard" real-time constraints which means that the computations are of no value if not completed on time. These characteristics, of course, affect the operating system. First, many of the features of "normal" time-sharing operating systems are not necessary. Secondly, the real-time constraints have extensive ramifications for the scheduling of processes. Further information on real-time operating systems can be found in Clark [Clar88a], Stankovic [Stan88], and Salkind [Salk88b].

HIC developed fortuitously from several activities in the Robotics Laboratory of the Courant Institute centered on the Utah/MIT dextrous hand [Jaco84]. One of the lab's projects involves the analysis of hierarchical control schemes for dextrous manipulation. This project generated the need for software suitable for these hierarchical control programs and the interprocess communication required for these programs. This work is an extension of our experience with the lab's Four Finger Manipulator, a four fingered

planar manipulator [Demm88] [Hor87] [Fehl87] and our expectations for the Utah/MIT hand. A parallel activity is GANGLIA which is a communication architecture for distributed robot controllers developed in the Lab [Clar88b]. GANGLIA encourages a highly stylized form of inter-process communication, suitable to much real-time communication, and the *periodic queues* of HIC have much the same form. The third activity is the installation of Condor for controlling the Utah/MIT hand. Condor [Nara87] was developed at MIT for the control of the hand and is a complete system from low level device and process control to convenient user interface. The Minimal Operating System (MOS) is a simple operating system within Condor that is the basis for much of HIC. In addition, HIC currently uses many of the low level routines from Condor.

HIC runs on multiple Motorola 68020 based micro-computers on a VME bus. Each processor is independent but communicates with the other processors via shared memory. There is a host Sun workstation that is used for program development and has complete access to the memory on the VME bus. The host is also used while HIC is running for debugging, user interaction, and supervisory control. Each of the micro-computers has a single background process and some number of periodic and/or asynchronous processes often called *events* in HIC. The background task has complete control of the processor when it is running but is preempted by the triggering of any of the events. An *event* is simply the invocation of a sequence of procedures. The events are strictly prioritized and at any time the highest priority event that has been triggered will be executing. The events are not processes in the normal sense, all the events on a processor share the same stack (the background task's stack) and therefore an event can not suspend execution except to allow a higher priority event to execute. That is, the events cannot *block* to wait for some condition to arise. Thus the events should fall into the "closed form" described above and once started an event should proceed expeditiously to its conclusion. An asynchronous event is triggered by an external interrupt, for instance, and runs to completion and does not run again unless it is triggered again. A periodic event is automatically rescheduled to be triggered by the clock each time is executed.

Section 2

Hierarchical Control

Hierarchical design is common in computer programs. The reason is two-fold. The first is for the user's/programmer's convenience. For instance, an operating system might present disk storage to the programmer as sequential and random access files, hiding the specific details of disk access and file maintenance. On top of this are developed text files, indexed files, relational databases, and so on. Thus the user or programmer is presented with "familiar" abstractions suitable for the task at hand. Secondly, hierarchical programming is good software engineering. If one considers a system as having the raw devices on the bottom and the application programs on the top then hierarchical programming represents vertical modularity and all the various advantages of modular programming apply. The specification for each step in the hierarchy can be precisely stated simplifying development. Changes in the underlying hardware need not propagate changes throughout the system but can be handled by the lower levels of the hierarchy. Programming modifications and optimizations can be applied at the appropriate level(s) again without disrupting the entire system.

All of this applies to robot control programs as well. Furthermore, our experience has shown that there is a similarity of structure in each level of the control hierarchy for robot manipulators, at least in the lower levels. HIC is an operating system meant to exploit the hierarchical structure and the similarity of the control levels. It is likely that similar structures would be found in control programs for robot arms, or walking robots, or other robot systems but the discussion in this document is focused on robot manipulators with particular emphasis on the Utah/MIT hand.

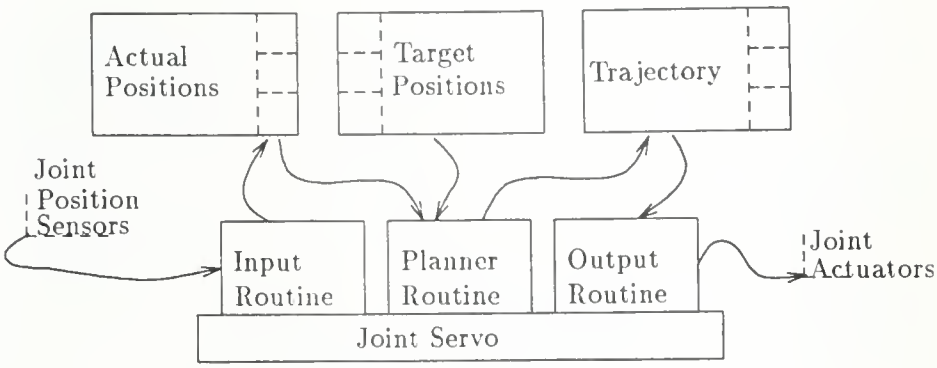


Figure 2.1: Simple Joint Position Servo

2.1 A Joint Servo

Consider a joint position servo, Figure 2.1 is a schematic illustration. This servo consists of three procedures. The *input procedure* obtains readings from the joints' position sensors and produces the *actual positions*. In general the input procedure may manipulate sensor readings, it might convert the raw sensor values to more meaningful units (radians for instance), or the readings may be filtered to remove noise, or some other manipulation to put the data in appropriate form for use by the other parts of the system. In the figure the input routine is shown placing the actual position values in a structure known as a *periodic data queue* or pdq. Pdq's are interesting structures that play an important role in RIC but at this point it is sufficient to understand that routines can *put* things (data aggregates) into them and other procedures can at a later time *get* access to those same things. So in this example the input routine places actual joint positions into the actual position pdq and the *planner routine*, the second procedure, will get those position values from the pdq. The planner procedure is invoked after the input routine and obtains the actual positions placed on the actual position pdq by the input routine and target or goal positions from the *target position* pdq. It applies appropriate control laws for the joints producing the trajectories to be made by each joint and places them on the *trajectory* pdq. finally, the *output routine* picks up the trajectories and commands the joint actuators to perform the corresponding motions. Target positions are not

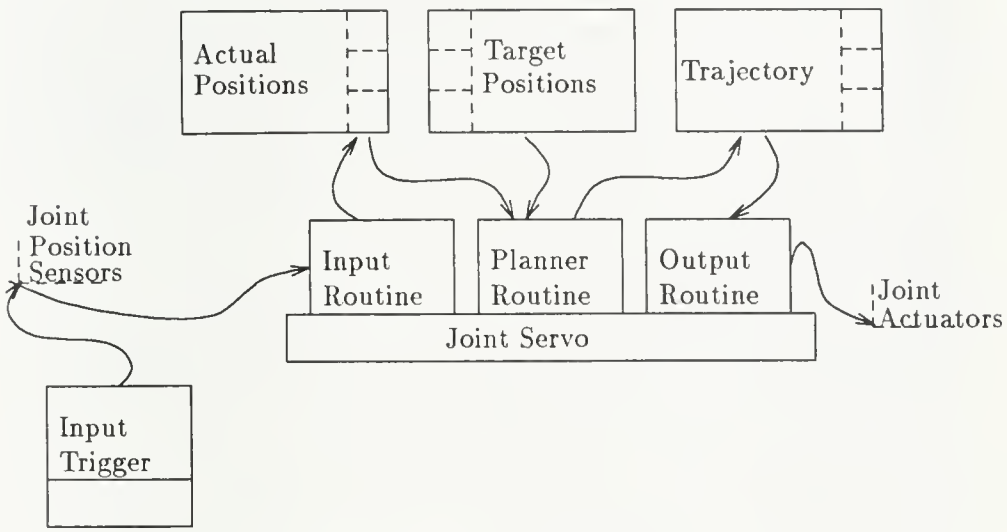


Figure 2.2: Handling a Complex I/O Device.

produced by this servo but are generated by some higher control level which places them on the target pdq. This sequence of three phases, input — plan — output, is repeated at a frequency appropriate for the devices being controlled. As we will see below, the three phase servo also appears in the higher levels of the hierarchy and is a central motif in the HIC model.

This example in isolation is rather simple but it illustrates the major components of HIC. The servo routines collectively are called alternately a *sequence* or an *event*. An event or sequence consists of an arbitrarily long list of procedures that are invoked sequentially when the event is triggered. An event can be triggered in one of three ways, it can be triggered by the background task or by a procedure in some other event, it can be triggered by the handler for a hardware interrupt, or it can be scheduled in a timer queue and triggered when the scheduled time arrives. This final method is typically used to trigger servo sequences. The joint servo is flagged as a periodic event so that when it completes execution the HIC system automatically schedules it for the next cycle. There are further refinements on sequences and their procedures but discussion of these details is postponed until Section 3.6.

It is important to remember though that events are not processes in the

normal multi-processing sense, in particular the procedures in a sequence can not block while waiting for I/O to complete. The HIC system itself does not provide support for or impose structure on the I/O devices so that I/O is left entirely up to the procedures involved. Our example servo assumes that obtaining the input values is not a time-consuming multi-stage process. This is often true, for instance, when the inputs are analog signals digitized by an analog-to-digital converter. Other devices may not be so simple. Suppose that a joint position device inputs its values by direct memory access and that to obtain values it were necessary to initiate the acquisition and then roughly 1 milli-second later the values would be available in memory. One milli-second is a long of time to busy-wait for the input to complete so our previous example might be modified as shown in Figure 2.2. Here another event has been added, the *input trigger*. The servo loop is scheduled by scheduling the trigger at the appropriate interval. When executed it initiates the input device and then 1 msec later the main body of the servo loop is triggered and the sequence proceeds as before. The main part of the servo loop could be triggered in several ways. The two events (the trigger and the body) could each be scheduled periodically with 1 msec interval in between their scheduled times. This is dangerous because HIC does not guarantee that events are executed when scheduled only that they will be executed sometime after they are scheduled depending on the relative priority of the event with other activities on the processor. Thus there is no assurance that there is enough time between the execution of the events for the input to complete. An alternative is to have the trigger event schedule the second event for 1 milli-second after it has initiated the input, this can ensure there is enough time for the input to complete. Yet another alternative could be used if the input device is able to interrupt the processor when the input is complete. This interrupt can be used to trigger the main part of the servo loop. Note in this case the main event is not scheduled with the timer but is directly triggered by the input device's interrupt. What method is most appropriate is a design choice based on the specifics of the system.

Periodic data queues are also shown in the example systems. These are message passing queues specially designed for periodically produced data such as that used by the servo loops. Each pdq is for a particular type of data, actual values, target values, or trajectories. The basic operation on pdq's are *put* and *get*. Data is put on a pdq, by the input routine for instance, and these values become the most recent values. A routine performing a get on a pdq gets a pointer to the most recent values put on the queue. In our example the pdq's behave simply as buffers since there is only one source

and one destination for the pdq's and they execute sequentially. However, in a system with multiple servo loops and supervisory programs running at various rates on multiple processors pdq's are an efficient method to make data available. In particular there are no timing or synchronization requirements between the various routines accessing a pdq. Data can be put on a pdq at any time and that data will become the most recent data on the queue. A get on a pdq returns a pointer to the most recent data on the queue and the data is accessible by the routine as long as it keeps the pointer, that is, the data will not be overwritten even though new data may be placed on the queue. All routines performing gets around the same time will receive pointers to the same data, the data is actually shared by the various processes. For this reason the data received via a get should be viewed as read only by the process receiving it.

2.2 Utah/MIT Hand Control

Finally, consider a more complete example, Figure 2.3. First of all, this example clearly shows the similarity of structure in the three levels of the hierarchy. This example is similar to the control hierarchy we propose to use on the Utah/MIT hand in our laboratory. This example is discussed in some detail both to illustrate how a control program can use HIC and to document the planned control scheme for the hand. At present the lowest level is implemented and operational as described and the finger and object level servo's are being implemented.

Raw joint servo. The raw joint servo is more complex than in the first example. The input is both the joint angles for the 16 joints and the tensions on the 32 tendons (two per joint). The input procedure does little manipulation of the input data. The joint angles are read as 12-bit signed integers and are converted to 16-bit integers and put in the actual pdq without further modification. The strain values are 12-bit signed integers, they are converted by a simple formula to torques at each joint, 16-bit values stored along with the angles.

Target values are supplied by the next level in the hierarchy, the finger servo. These values correspond directly to the actual values, there are 16 desired joint and 16 desired torques both in the same units as the corresponding actual values. The control parameters are parameters for the planner routine. The planner takes as its inputs the most recent actual

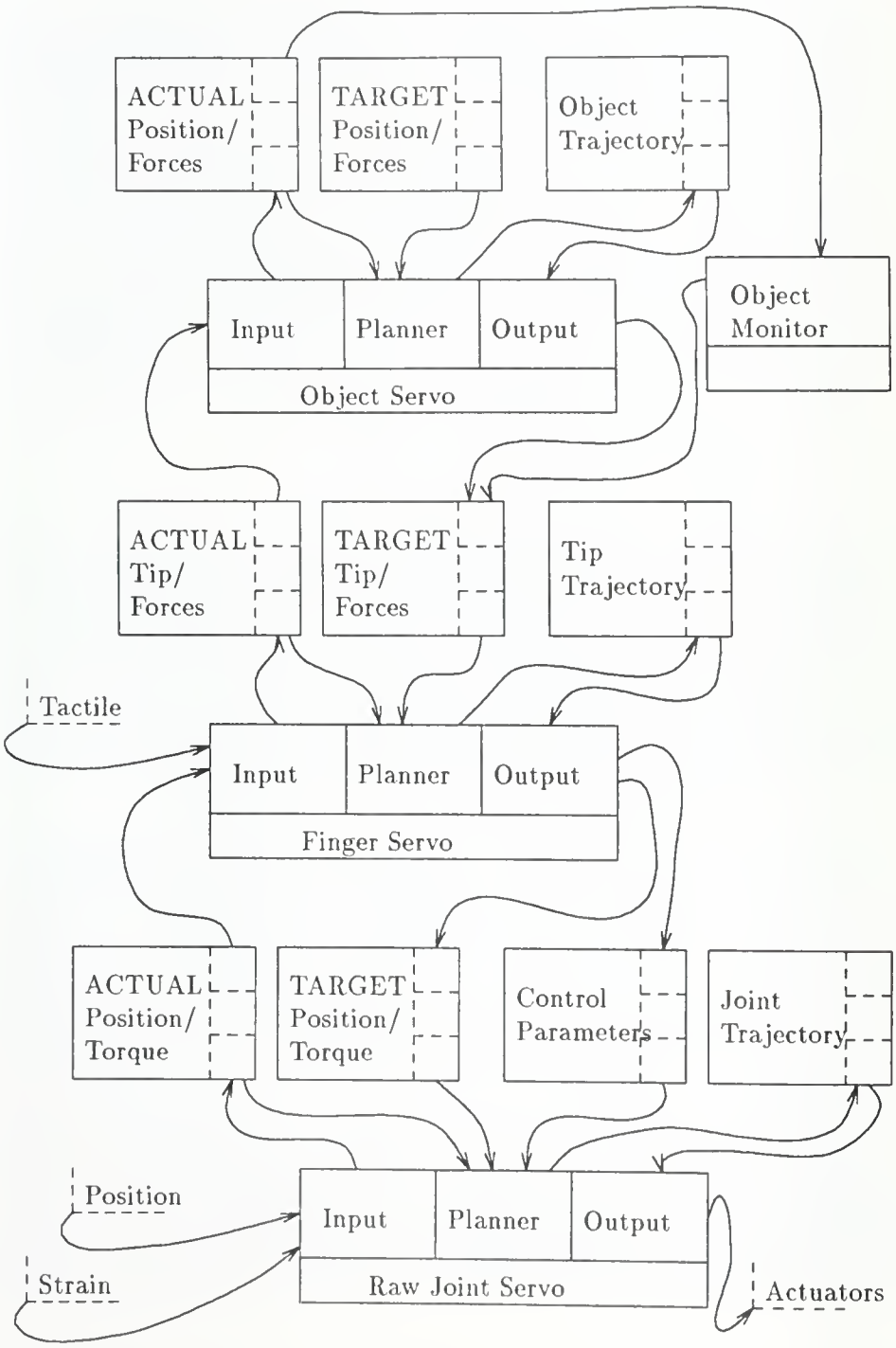


Figure 2.3: A Three Level Hierarchy.

values and the most recent target values and applies an amalgam of force control and position control on the individual joints using the most recent control parameters. Gains for both position feedback and force feedback (for the individual joints) are obtained from the control parameters. Typically for any one joint one of the two gains is zero, thus reducing the amalgamated force/position control simply to force or position control. In the figure, control parameters are shown as comming from the finger servo, actually it is likely that they may be set by any of the higher levels of control. Finally, the planner produces trajectories (16-bit integers) for the joint actuators which are applied by the output procedure without conversion (except for 16-bit to 12-bit integers). For reasons of speed all values in the joint servo are integer and the units are those used by the analog-to-digital and digital-to-analog converters, hence the name "raw joint servo".

In this example it is easier to see the value of periodic data queues. The pdq's act as buffers between the control levels both in that they hold data as it passes between levels and that they smooth problems presented by the fact that the servos at different levels may be operating at different rates and may reside on different processors. It is likely that there will be nearly an order of magnitude difference in the rates of the joint servo and the finger servo and they are also likely to execute on different processors and yet a pdq will, to any request for data, always return a "reasonable" set of values (i.e. the most recent values available). Debugging procedures and system monitoring procedures can make use of pdq's also. Although not shown in the figure, the pdq's can be accessed by any procedure in the system so that a monitor process on the host computer might monitor or record the joint positions by periodically getting a "snapshot" from the actual pdq. Any process can also perform puts on a pdq so that a monitor or debugging process could override the finger servo and supply target positions directly to the joint servo. It is for these reasons that there is a joint trajectory pdq which on the surface seems a bit superfluous since it connects only the joint planner and the joint output routines which are always run sequentially (a simple buffer could be used instead). But putting the trajectories in a pdq allows a process to monitor them. In the example the finger servo provides the control parameters for the joint servo, a likely alternative is that a task level process might be responsible for the joint level control parameters. Pdq's make all of this possible with little interference to the basic system.

Pdq's are a little more complicated to use than presented so far. To the basic access procedures, *put* and *get* there are two complimentary procedures *reserve* and *unget*. A process that wishes to put values on a pdq must first

reserve a buffer (using the reserve call) the data is placed in the buffer then the buffer is put on the queue. Similarly when reading from a pdq a process gets a buffer (actually a pointer to the buffer) and uses the data as needed. When done examining the buffer an unget is performed to terminate access to the buffer.

Finger servo. As remarked earlier, the finger servo is similar in structure to the joint servo. There are three procedures input, planning, and output. Input to the finger servo consists mainly of the joint positions from the lower level joint servo. Forward kinematics for the fingers are applied to the joint positions to obtain the positions of the finger tips in Cartesian coordinates. Using the torques on the individual joints, an estimate of the forces being applied by the finger tip can be obtained. Only an estimate is possible because the joint torques alone are not enough to determine the finger tip forces. We have plans to add tactile sensors to the finger tips, and input from these sensors would be appropriate at this point. These sensors will allow a more accurate determination of the forces on the fingers. The results are placed in a pdq for the actual finger tip positions and finger tip forces. The planning procedure picks up the actual position and force values and the corresponding targets. Then using an appropriate control formula produces trajectories for the output procedure. It is up to the output routine then to apply the inverse kinematics and produce target values for the joint servo.

Note that in Figure 2.3 there is no parameter pdq for the finger servo (or the object servo). This is more a matter of aesthetics than design. Any control scheme is likely to have operation parameters and if these change while the system is in operation then a pdq should be used to hold the parameter values. But, adding these parameter pdq's to the figure complicates the picture without significantly increasing the illustrative value. So the parameter pdq's have been left out of the figure.

Object servo. The object level servo is not so intuitive. When the fingers have a fixed grip on a rigid object then there is a simple coordinate transformation between the position of the fingers and the position and orientation of the object. It is possible by a linear transformation to deduce the external forces applied to the object from the forces sensed at the finger tips (the points of contact with the object) [Demm88]. Note that not all forces sensed by the fingers are due to external forces since the fingers must apply force to maintain a grasp of the object.

In a very real sense the entire system (joints, fingers, and object) become a single finger. Consider a person writing with a pencil. The pencil is gripped firmly and the forces transmitted through the pencil to the fingers is used, in part, to determine the nature of the writing surface and the amount of force to apply. Manipulation of an object or tool in which the grasp is fixed and a single control scheme is used on the tool is called a *homogeneous manipulation* or *homogeneous task* [Demm88]. Homogeneous tasks can be strung together, with appropriate transitions, to make more complex tasks. Thus writing can be decomposed into: grasping the pencil (not homogeneous); transporting it to the paper (a homogeneous task); applying the pencil to the paper (homogeneous); and making marks on the paper via a sequence of stokes (homogeneous tasks). This highlights an important feature of the HIC system, namely that the procedures in an event are "pluggable modules". In the end, as far as the HIC system is concerned, the object level planner is just a pointer to the actual procedure and a higher level process can change the pointer as desired. Thus there may be a planning procedure for maintaining a grip while the hand is in free motion, another planner for bringing the pencil into contact with the paper, and perhaps different planners for the different types of strokes made while writing. So managing the writing task is now largely a matter of switching from one planning procedure to another at the appropriate times and providing the appropriate parameters. The planning modules are not the only ones that are pluggable, any procedure in a sequence can be replaced in the same way, so if it were useful, the input or output procedures can be changed as needed during a task. Although the example sequences in this paper tend to have the same form of input — planner — output, the HIC system itself if more general. The reader should remember that sequences contain arbitrarily long lists of procedures and that each procedure is pluggable.

Now back to the example in Figure 2.3. During an homogeneous manipulation the object servo input procedure takes the actual positions of the finger tips and calculates the position and orientation of the object. Also the forces at the finger tips are used to calculate the external forces on the object. The planner then compares these with the target values and calculates an appropriate trajectory for the object. The output procedure transforms the object trajectory into targets for the finger servo.

Higher level. Where do the object's targets come from? Higher level processes. The higher level processes are less likely to fit the "closed form" described in the very beginning of the report and therefore less well suited for

HIC. They will run under more conventional real-time operating systems, such as Sage [Salk88a] or perhaps on the host computer. However, these control levels will have access to the pdq's, in fact the pdq's will be the primary interface between the higher level control and the low level servo loops.

2.3 The NASA/NBS Standard Reference Model.

It is interesting to compare HIC to the reference model for telerobot control systems developed by the NASA and the National Bureau of Standards. This model, known as NASREM, specifies the control system for the NASA Space Station IOC Flight Telerobot Servicer which is the conglomeration of robots and service bays for the repair of satellites and other spacecraft on the proposed space station [Albu87]. NASREM is a particular application of a general architecture for hierarchical control systems described in *Hierarchical Control for Robots and Teleoperators* [Albu85]. Figure 2.4 shows a schematic of the NASREM model.

The scopes of the two models are quite different. NASREM is a model for the complete control hierarchy for a multi-robot system. In the figure one can see the model extends from the joint servo at level one to the service bay control at level five which allocates the various robots in a single service bay to the tasks at hand to the mission level which allocates the spacecraft and repair problems to the various service bays. The global memory, which contains information about the entire hierarchy is conceptually a single memory but actually is spread among the various components of the system on various media and may, in fact, reside partially in earth bound components. HIC on the other hand is much more limited in scope. HIC is a model structure for the lowest levels of robot control, the three phase servo loops, which makes few assumptions about the higher levels. HIC assumes that the higher levels can use periodic data queues for communication purposes but that the higher control levels are probably unsuited for HIC scheduler (because they do not fit the "closed form" described in the Introduction). In terms of distribution of the system HIC basically assumes that the components of the control system share a common address space on a single bus. HIC, though, is compatible with GANGLIA which is a communication system intended to allow the low levels of a robot controller to be distributed within the robot being controlled [Clar88b]. The HIC model is also what might be termed a "minimalist" model. The hierarchy of three phase servos is put forward

as a model for the core of a control system with the expectation that additional components will be added to match the requirements or subtleties of a particular system.

Referring to Figure 2.4 one can see that NASREM has three major components the global memory, the control hierarchy, and the operator interface. The global memory, as mentioned above, holds all the shared information about the system. It is a distributed memory existing in various locations and on various media. Conceptually the memory is a uniform memory with the data items accessed symbolically. Memory in a HIC system is all on a common bus and accessible by all the components of the system. Pdq's provide a symbolic method for accessing the shared data. In NASREM, the operator interface interacts with all levels of the system. It allows a human operator to monitor operation of any level of the hierarchy and to intercede at any of the levels. HIC does not explicitly include an operator interface but the pdq's provide the necessary interface. Since the pdq's are globally readable monitor processes can be added with very little disruption to the control processes. And likewise, any process can put an item on a queue so an operator input procedure can provide target or parameter values for any level of control. This latter case though will most likely require some direct coordination between the operator's input process and the process that normally provides the targets or parameters. The program, rjedit, (see Section 5.3) provides a simple operator interface to the joint servo used with the Utah/MIT hand.

In NASREM, the controller is partitioned both vertically (the hierarchy) and horizontally, much as the HIC model. Each level of the hierarchy has a *Sensory Processing* module (called the "G" module, i.e. G1, G2, etc.) A G module takes sensory information, primarily from the next lower level, produces the necessary information for its level of control and gives it to the world model module which places it in the global memory. This is identical to the function of the input phase of the HIC model.

The *task decomposition* (or "H") modules correspond directly to the planning and output phases of the HIC model. Within an H module are *planners* which do the planning appropriate for its level and *executors* which carry out the plan. For instance, H5 (the service bay task decomposition module) may allocate a job among several manipulators by assigning a planner for each of the manipulators. Each planner would be responsible for the task assigned to its manipulator and would use executors as needed to carry out the plan.

The *world model* (or "M") modules do not have such direct counterparts

in the HIC model. An M module is responsible for maintaining its level's view of the "world". It is also responsible for interpreting the world model for its own planner as well as other planners in the hierarchy. It takes the processed sensory data from the G module at its level and updates the global memory. It also produces predictions of expected sensory input for the corresponding G module. For the planners, the M module will answer two types of questions "What is?" questions and "What if?" questions. "What is?" questions are direct queries of the state of the world. For instance, a planner might ask "What is the position of joint 3?" or "Has the manipulator completed the move?" (i.e. "What is the state of the manipulator?"). Planners may also make hypothetical "What if?" queries like "What if the arm were moved to position X?". The M module is also charged with evaluating the current state and predicting future problems. So an M module might inform its planner that maintaining the current configuration is costly in terms the energy consumed or that continuing on the current path will lead to a collision.

One might say that the pdq's associated with a servo in the HIC model (actual values, target values, parameters, and trajectories) collectively correspond to an M module in NASREM. These *do* represent the "world" as seen by the servo and provide the basis for answers to "What is?" queries. But there are no formal provisions for answering "What if?" questions. Answers to these sorts of questions are diffused throughout the code. So instead of asking the finger servo "What if you are commanded to go to point Y?" the reverse kinematics procedure is invoked with Y as its argument and the procedure will return with an error if Y is illegal and the proper joint positions if it is valid.

The servo structure is not as prominent in NASREM as it is in the HIC model. The planner at any level has a *planning horizon* which how far into the future it can plan. As would be expected, the planning horizon increases for each level in the hierarchy. So it is clear that the planner will be executed periodically but the interval is not necessarily fixed, as it is in HIC. In an example system, the NASREM authors do, however, have the lowest levels of the hierarchy execute at regular intervals, which corresponds with the HIC model. The relationship between the execution of the modules in a level in NASREM is not made explicit, as it is in HIC.

The first two levels of the NASREM hierarchy are very similar to the joint and finger level servo in the Utah/MIT hand controller. Level 1 (the servo level) servos the robots' joints in the joint's coordinates, just as in the hand's controller. NASREM has the level 1 perform the kinematic con-

versions between the joint coordinates and the coordinate frame of choice for the robot. In the hand's controller this is assigned to the finger level servo. The primitive level of NASREM servos the end effector in the preferred coordinates, just as in the hand's system. In NASREM this level also computes inertial dynamics for the robot. Currently, we treat the hand as quasi-static and do not compute dynamics. The finger servo would be the appropriate place to include dynamic factors.

"E-move" in Figure 2.4 stands for *elementary move*. From the NASREM technical report:

> E-moves are typically defined in terms of motion of the subsystem being controlled (i.e., transporter, manipulator, camera platform, etc.) through a space defined by a convenient coordinate system. E-move commands may consist of symbolic names of elementary movements, or may be expressed as keyframe descriptions of desired relationships to be achieved between system state variables. E-moves are decomposed into strings of intermediate poses which define motion pathways that have checked for clearance with potential obstacles, and which avoid kinematic singularities. [Albu87]

This description is quite general because the model covers many types of robots in many situations. Applied to the Utah/MIT hand this description covers the functions of the object servo described above. So the third level of NASREM corresponds closely to the third level in the hand's controller.

The task level (level 4) decomposes an assigned task into E-moves to be given to lower levels. This is just what is expected from the higher control levels in the hands controllers. This level and higher levels are deemed to be outside the purview of the HIC system and will reside either in the host processor or on other real-time systems (for example a Sage [Salk88a] system). The top levels of the NASREM system are responsible for managing multi-robot systems and the multiple service bays on the space station. These are clearly beyond the scope of HIC and this report.

One final point of comparison is interesting. The global memory of the NASREM system is *open* to all components and levels of the system and the world model at any level can be queried by *any* of the planners. Likewise, the pdq's in a HIC system are open to all components of the system. The point being that the systems are organized hierarchically but information flow is *not* restricted to the hierarchical structure. It is tempting when considering a hierarchical system to see all information flow following the lines of the

hierarchy (i.e. within one level of the hierarchy or between neighboring levels). The vast majority of information flow should be along the lines of the hierarchy but there are always cases where it is easier or more efficient to "go outside the" levels in the hierarchy to access data. For example consider a monitor process whose task it is to identify illegal positions (joint singularities, finger collisions, impossible object positions, etc.) before they occur. Such a task might well need access to the actual and target positions for all each of the levels of control and it might affect the operation of all of the planners.

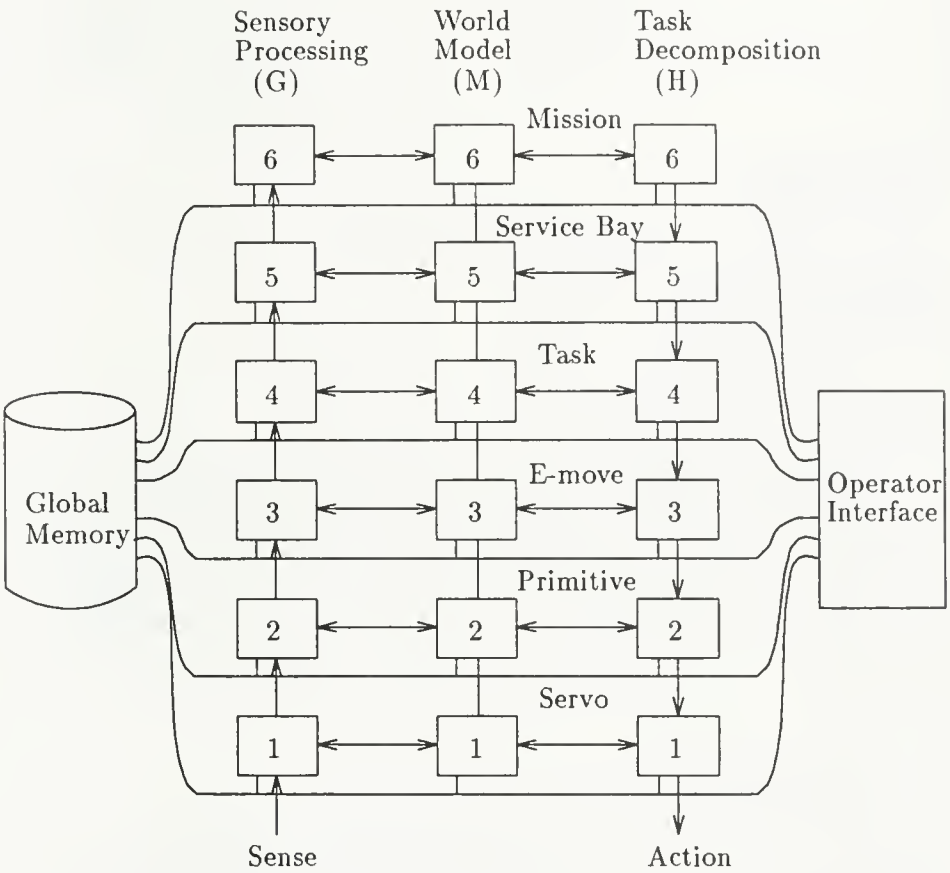


Figure 2.4: NASA/NBS Standard Reference Model.

Section 3

Overview

The next few sections of this report discuss the implementation and use of HIC. First the hardware on which the system is implemented is presented (Figure 3.1). This is the system used to control the Utah/MIT hand. Then the HIC software implementation is described in detail. In the following section the HIC library routines are described. Then example programs using HIC are described. These are various program developed for the Utah/MIT hand.

3.1 Hardware

The heart of the system for controlling the hand consists of four Motorola 68020 processor boards on the VME bus. The boards are manufactured by Ironics (IV-3201) [Iron86] and each has a 68020 processor and 1 megabyte of memory. The memory is dual-ported so it is accessible both by the processor and by other VME bus masters (i.e. the other processors). The primary form of interprocessor communication is the *mailbox interrupt*. Writing to a known location in a processors local address space causes an interrupt to the processor [Iron86]. Condor implements a remote procedure call protocol using the Ironics mailbox interrupt, by which any of 256 previously specified procedures can be invoked, the procedure is given a single word argument and can return a single word value. Each processor has a single interval timer. The processors are numbered from 0 to 3 and their memories appear on the VME bus at addresses 0xb00000, 0xc00000, 0xd00000, and 0xe00000 respectively. Thus for each processor its own memory resides at address 0x000000 – 0x0fffff and the other processors' memories reside at the above

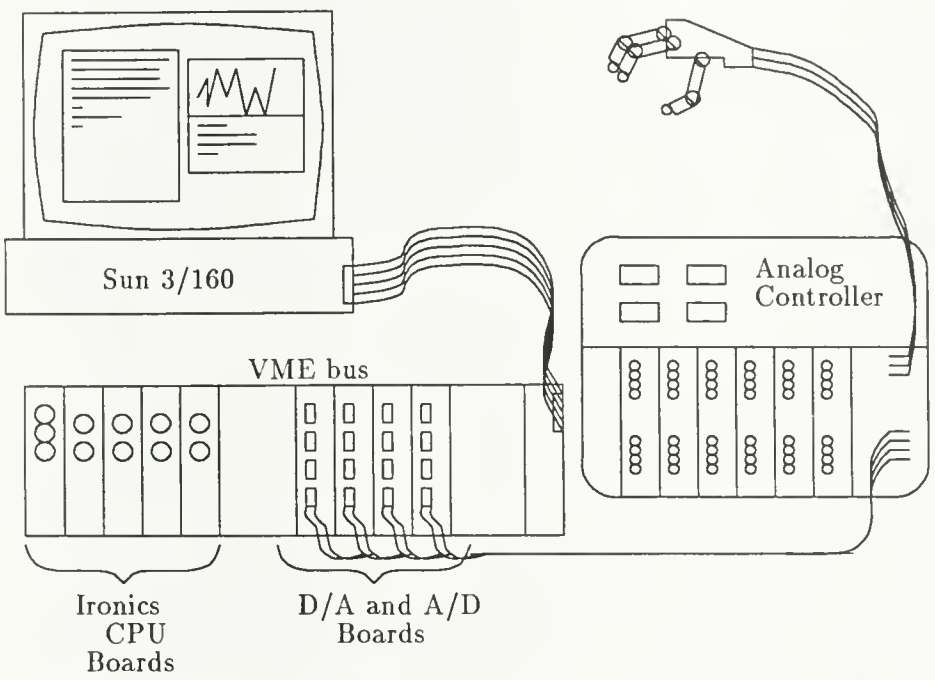


Figure 3.1: Control Hardware.

locations. A processor can only refer to its own memory at the local address, not at the global or VME bus address.

Another Ironics board, the IV-3273 System Controller is the VME bus arbitrator which means that it arbitrates the requests by bus masters (i.e. the processor boards) for control of the bus for data transfers. The IV-3273 also contains the connections to the console terminal for the processors. Unlike most single board computers the Ironics boards do not have individual console connections, they must share the single console on the system controller board. In practice only one of the processors (processor 0) uses the console. Pseudo terminal connections to the host computer, based on those provided by Condor, can be used as console terminals for the processors.

A Sun 3/160 is the host system for the hand controller. It is connected to the VME bus via a bus connector from HVE Engineering (HVE Synergist III) [HVE86]. This makes all of the memory on the VME bus visible to the Sun. Programs for the controller are compiled on the Sun using the Sun's own compiler and are downloaded directly to the appropriate proces-

sor's memory. The processor can then be started via a mailbox interrupt. Once running, the controller programs and the host can communicate via the shared memory, on the VME bus. Condor also allows the controller's processors to cause mailbox interrupts on the host and vice versa. In particular, the host has access to HIC periodic data queues on the controller which allow it to monitor and interact with the processes. Another feature provided by Condor is support for the GNU debugger (gdb) [Stal87] which is a source level debugger for programs written in C running on the controller's processors.

The hand's actuators and sensors are connected to an analog controller. The controller contains analog servos for position, velocity, and torque. It provides as output analog signals for current position, current velocity, and the tension on each of the tendons (two per joint). It accepts as input analog signals for the desired position, position gain, velocity gain, and gains for the tendon tension. An optional mode allows direct access to the actuators, bypassing the analog servo loops. The analog controller is connected to the digital controller via digital-to-analog and analog-to-digital converts on the VME bus. These are manufactured by Data Translation (models DT1406 & DT1401) [Tran86a] [Tran86b].

Multi-processor issues. HIC is designed to run on multiple processors sharing memory. In particular, periodic data queues are accessible across processor boundaries. There are also inter-processor mailbox interrupts in Condor which underlies HIC. The Sun host processor communicates with the HIC system via both pdq's and mailbox interrupts. The processors though run independently. Each processor has its own stack, timer, priority list, scheduled list, and set of events which are statically assigned to the processors. There are no built-in provisions in HIC for load balancing or for events to migrate from one processor to another.

The processor boards on which HIC has been developed do not have any form of memory management. This isolates, to some extent, the individual processors and their local memory. One result of this is that a pdq and all of its PBUF's must reside in the local memory of one of the processors, a mild restriction.

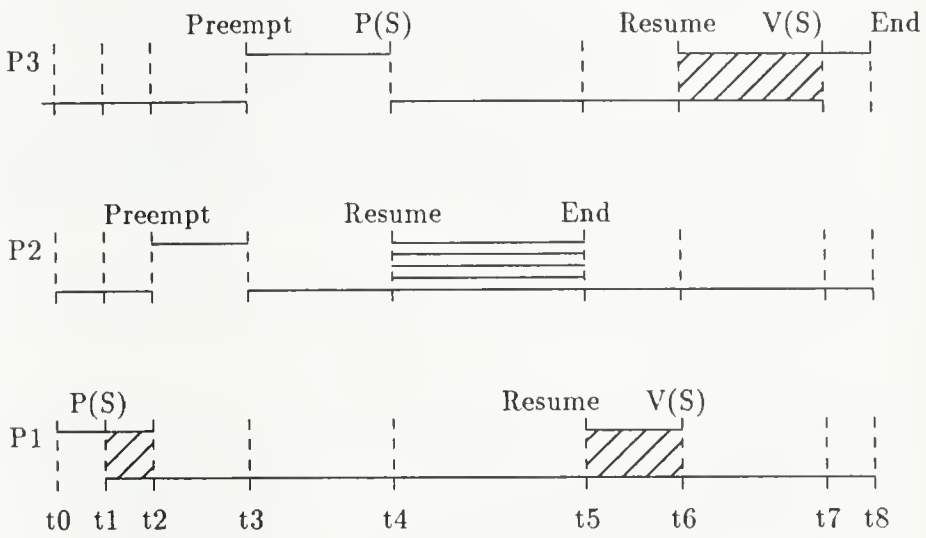


Figure 3.2: Example of priority inversion.

3.2 Real-time Multiprocessing

The HIC operating system does not provide multi-processing facilities in the normal sense. The most striking difference is that there is only a single stack per processor with the immediate consequence that processes can not block since a blocked process prevents any of the preempted processes below it on the stack from continuing execution. Processes can be preempted however. If a higher priority event (i.e. process) is triggered then the current process is suspended and the higher priority process is allowed to proceed. And since the new event will not block, it will run to completion, unless, of course, an event with still higher priority is triggered. It is easy to see that this ensures that at any point the highest priority readied event is running (on a per processor basis). This predictable behavior is very desirable in real-time operating systems. A problem which HIC avoids by prohibiting blocking is the problem of *priority inversion.*

Consider an example (Figure 3.2) of three processes, $P1$ (low priority), $P2$, and $P3$ (high priority) and a shared resource (represented by the lock S) in a normal multi-tasking system with simple priority scheduling.

- At time $t0$, $P1$ is executing.

- At $t1$, $P1$ requests and obtains the lock S and enters its critical section (represented by the angled hash lines).
- At $t2$, $P2$ preempts $P1$ and executes until $t3$ when $P3$ preempts $P2$.
- At $t4$, $P3$ requests S and blocks because $P1$ is holding S. When $P3$ blocks $P2$ is resumed since it is the highest priority runnable process. This is the priority inversion (represented by the horizontal hash lines).
- At $t5$, $P2$ completes and $P1$ is resumed, in the midst of its critical section which completes at $t6$.
- At $t6$ when $P1$ completes its critical section, $P3$ is given the lock S and enters its critical section which continues until $t7$ when S is released and $P3$ continues.

During the interval from $t4$ to $t5$, $P3$ is waiting for $P1$ to complete its critical section but $P2$ is executing. This is the *priority inversion* since $P3$ is now essentially preempted by $P2$ which is of lower priority. Note that the interval of $t4$ to $t5$ is arbitrarily long. In a real-time system with hard deadlines this is intolerable since a time critical process can be blocked for unbounded intervals by lower priority processes. $P3$ is also blocked during the interval from $t5$ to $t6$ but this is an unavoidable side affect of resource sharing. In addition, the length of this delay can be bounded since in a well designed system the critical sections are short and well known so that the designer of $P3$ can take them into account.

There are ways to avoid priority inversion short of the draconian method of prohibiting blocking, taken by HIC. A straight forward technique is *priority inheritance* in which a process holding a resource inherits the priority of the highest priority task blocked on that resource. Figure 3.3 shows the same example with priority inheritance. Note that the same amount of work is performed in the same time but that when $P3$ blocks at $t4$, $P1$ inherits the priority of $P3$ and therefore takes precedence over $P2$. $P1$ is then able to complete its critical section and $P3$ is able to continue as soon as is possible. Since a process is never blocked for a period longer than a critical section it is possible to predict the maximum amount of time that a process will block (assuming that the number of critical sections is small and each critical section is bounded in execution time).

HIC takes the approach of prohibiting blocking for efficiency reasons. It is able to run with a single stack per processor. Since there is only one stack, there is very little state information that needs to be saved during a context switch. In normal circumstances when an event completes it simply returns to the scheduler which searchs for the next highest event that is

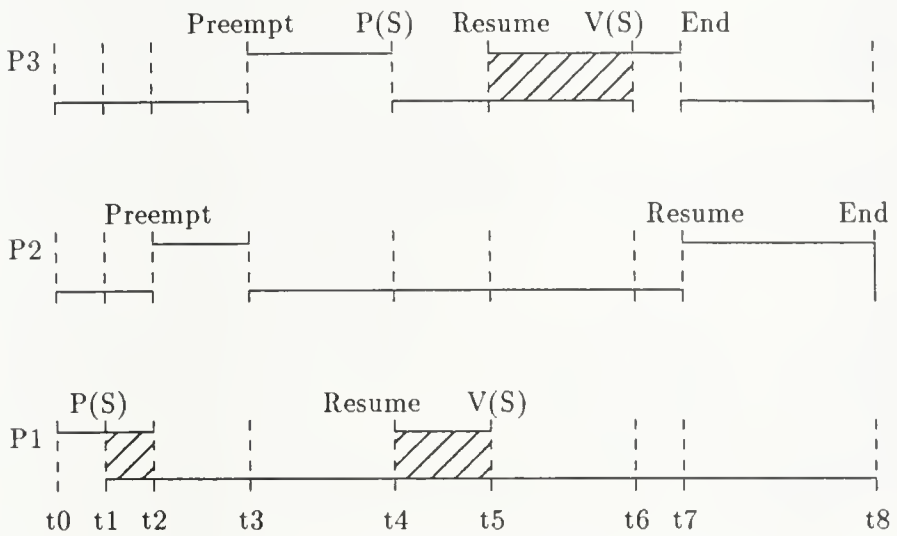


Figure 3.3: Same example with priority inheritance.

ready and invokes it with a subroutine call. If an event of lower priority is triggered either by the current process itself or by an interrupt procedure, the event is simply marked as ready and nothing else need be done. An event of higher priority than the current event can be triggered in two ways, the current event can trigger it in which case the event is immediately invoked with only little more overhead than a subroutine call or the event can be triggered by an interrupt handler where the overhead is the same but with the added overhead of the interrupt invocation. Why is it reasonable to prohibit blocking in HIC? The answer is that the lowest levels of robot control hierarchies are usually servo loops which can follow the HIC form of *input — compute — output* described in the beginning of this report. HIC's approach to resource allocation is that a process must have all of the resources required allocated *before* it is initiated. And that the triggering of an event is essentially a signal that all resources for a process are available. This is not quite true though. Processes in HIC are allowed to access periodic data queues at any time. This is permitted because the pdq's are designed to always return something "reasonable" without blocking. In the case of getting from a pdq "reasonable" means that the most recently available values are returned (possibly the same values as obtained by the last access)

```
/*
   EVENT the event structure.
 */
typedef
struct event
{
  struct event* next_scheduled_event;
  struct event* next_priority_event;
  unsigned long flags;
  int           id;
  char*         name;
  int           dispatch_count;
  int           priority;
  int           rank;
  TIME          scheduled_time;
  TIME          nominal_time;
  int           interval;
  PROC*         procedure_list;
 }EVENT;
```

Figure 3.4: Event structure (EVENT).

which for servo loops is often acceptable and preferable to having the loop block while waiting for new values. In the case of reserving a buffer to put something in a queue the pdq will return a buffer if one is available otherwise a null pointer is returned. To ensure that empty buffers are always available for processes reserving buffers a sufficient number of buffers must be allocated when the system is designed.

3.3 Events

Figures 3.4 and 3.5 show the C structures for HIC events and procedures respectively. Each event in the system has an associated structure and each event has a list of procedures to be invoked when the event is executed.

`Next_scheduled_event` and `next_priority_event` are pointers for the two lists which contain events. These are described in more detail below.

`Flags` contains flags associated with the event. One of the flags, `EVENT_SCHEDULED`, is set if the event has been scheduled in the timer list (see below). Another `EVENT_READY` is set if the event is ready to execute and `EVENT_ACTIVE` indicates the event is executing or has been preempted (i.e.

that the event is holding space on the stack). EVENT_PERIODIC indicates the event is a periodic event and should be automatically rescheduled. EVENT_PERMITTED is set to indicate the event can be executed, otherwise execution is inhibited.

Id is a system wide unique identifier for the event. If the event must be "well known" an id can be assigned by the user when it is created otherwise the system assigns a unique id.

Name is an ASCII string naming the event for debugging purposes.

Dispatch_count is a counter incremented each time the event is dispatched, for instrumentation purposes.

Priority and rank make up the priority of the event. Higher values indicate higher priority. Priority is the nominal priority which can be assigned by the user when the event is created and falls in the range 1 to 99. If the event is periodic the system will assign a standard priority based on the periodic rate, if desired. If all periodic events are assigned the standard priorities then they will be prioritized in rate monotonic order (more frequent events have higher priority).

The implementation of HIC requires that each event actually have unique priority (this is for efficiency reasons in the scheduler). When an event is entered into the system it is assigned a unique rank based on its nominal priority, priority, this rank is actually used by the system when comparing priorities. If two events are assigned the priority 20, for example, the first one will have rank 2099 and the second will have rank 2098. An unfortunate consequence of this restriction is that events with the same priority will not exhibit "round-robin" behavior as might be expected. That is, if an event is triggered that has the same priority as the currently executing event it is in general unknown whether the triggered event will preempt the current event (if it happens to have higher rank) or will it be executed after the current event completes (if it has lower rank). Since the distinction between the priority and the rank of an event is just a technical matter of implementation we will use the more meaningful phrase "priority of an event" and leave it to the reader to remember the subtle distinction between priority and rank.

Event priorities are essentially static. It is possible to change the event priority list as discussed below but to do this one must "step outside" the HIC system and this cannot be done "on the fly".

Scheduled_time, nominal_time, and interval are used to schedule events in a timer queue and reschedule periodic events. An event can be scheduled to be triggered at a particular time, while it is inserted in the list

of scheduled events `scheduled_time` holds the time is should be triggered.

When a periodic event completes execution it is rescheduled at the time `nominal_time` plus `interval` automatically and then `nominal_time` is set to this new value. The purpose of `nominal_time` is to help detect system overload and missed deadlines. If the processor is so overloaded that a periodic event misses a complete cycle then when it is rescheduled the time (i.e. `nominal_time + interval`) will be less than the current (in other words it will be scheduled in the past). HIC does not allow events to be scheduled in the past so it will schedule the event to be executed at the current time. In this situation, `nominal_time` and `scheduled_time` will differ. This is an indication that the processor is falling behind. HIC, itself, does not attempt to correct the problem. That is left to the user.

`Procedure_list` is the head of the list of procedures to be performed when the event is executed. The basic operation is that the procedures are executed in the sequence presented however as we will see below there can be exceptions.

An event is actually triggered by calling `hic_dispatch()` with a pointer to the `EVENT` structure for the event. If the event is of higher priority than the currently executing event then `hic_dispatch()` invokes the new event directly otherwise the event's ready flag is set and the event will be invoked in the proper order.

3.4 Procedures

Figure 3.5 shows the structure for HIC procedures.

`Next` points to the next procedure in the list of procedures in this event.

`Id` and `name` are similar to the same fields in the `EVENT` structure. `Id` is an unique identifier which can be assigned automatically and `name` is an ASCII name for debugging purposes.

`Invocation_count` is incremented each time the procedure is executed. It is for instrumentation purposes.

`Count`, `reset`, and `initial` provide some scheduling capability internal to an event. When a procedure is created `count` is given the value `initial`. As the HIC scheduler steps through the list of procedures for an event it decrements `count` and if it is greater than zero the corresponding function is *not* invoked. If `count` is zero or negative then the function is invoked and and `count` is set to `reset`.

An example of where this might be used is for a safety or monitor pro-

```
/*
   PROC the procedure structure.
 */
typedef
struct proc
{
   struct proc*  next;
   int           id;
   char*         name;
   int           invocation_count;
   int           count;
   int           reset;
   int           initial;
   void          (*entry)();
   void*         argument;
  }PROC;
```

Figure 3.5: Procedure structure (PROC).

cedure associated with a servo event. Suppose that the monitor procedure need be executed only every fifth time the servo is invoked. Then `reset` should be set to five. As another example suppose that input values for a servo loop are to be sampled 300 times a second but to reduce the effects of noise on the inputs three samples are averaged before they are given to the planning procedure. Then the servo event should be scheduled every 1/300 of a second and `reset` for the input procedure is set to one so it is invoked every time the servo is triggered. For the planning procedure and output procedures `reset` is set to three so they are invoked only every third time.

`Entry` points to the C function to be invoked for this procedure. When it is called it is passed the pointer, `argument`.

3.5 Dispatching and Scheduling

There are two lists for events, the *priority list* and the *scheduled list.* The priority list is a linked list of events in priority order with the head of the list being the highest priority event. Events are dispatched from the priority list. Figure 3.6 shows how the dispatcher and the priority list interact. The list is implemented as a linked list but is shown here as an array to simplify the figure. The events are in priority order with event 1 being the highest priority, and the events are represented by their states, Active (in the midst

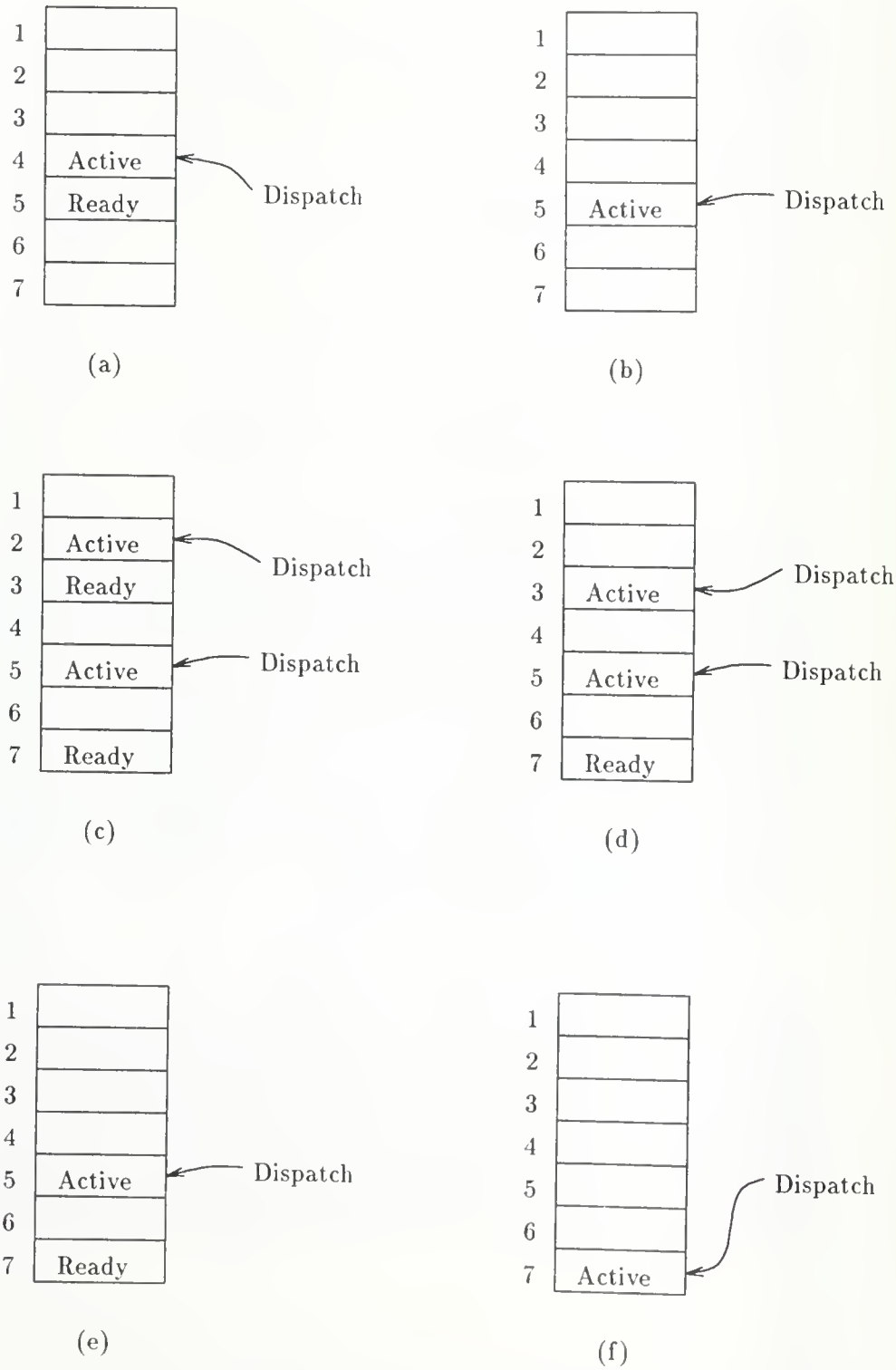


Figure 3.6: Priority list and the dispatcher.

of execution and holding space on the stack), Ready (ready to execute), and blank (not ready or idle).

Priority list. In Figure 3.6a there is one instance of the dispatcher with event 4 currently active and event 5 ready to execute. In Figure 3.6b event 4 has completed and the dispatcher has move on to event 5. While event 5 is executing the system timer interrupts, searches the list of scheduled processes, and finds three events whose time has expired (2, 3, and 7). These events are readied and since event 2 is higher in priority than the current event, 5, a second instance of the dispatcher is invoked on event 2. This is the state of the system in Figure 3.6c. When event 2 completes its dispatcher (the second instance) moves on to event 3 (3.6d). Then when event 3 completes the second dispatcher searches down the list for another ready process but it encounters the active process 5 which causes the second dispatcher to exit returning eventually to the first dispatcher which continues executing event 5 (3.6e). Finally, when 5 completes the dispatcher finds event 7 ready and executes it (3.6f). If the dispatcher falls of the end of the list it performs a phantom event, the background task. In general, any number of instances of the dispatcher can be running and for each dispatcher there is one active event. When an event exits its dispatcher searches down the priority list until either a ready event is encountered, which is then executed, or an active event is encountered in which case the dispatcher exits.

Notice that every active process is holding some of the processor's stack, these processes cannot be removed from the system until they complete their execution and release the stack space they hold. This makes changing the priority list a risky proposition without thorough knowledge of the states of all the events. However, an event while executing can be sure that there are no events of higher priority active, so if an event is sure it won't be preempted it can change the priority list between itself and the top of the list. The proper place to reconfigure the priority list is in the background task. The background task can be sure that no event in the list is active so the entire list can be modified.

What happens if an event is triggered while it is active? If the dispatcher, `hic_dispatch()`, is called with the current event as its argument it simply sets the ready flag, `EVENT_READY`, since the priority of the triggered event is *not greater* than the priority of the current event (since the triggered event *is* the current event). When the current event completes execution the dispatcher checks its ready flag and if it is set the event is invoked again. Thus a malicious or malformed event could lock out all lower priority events

```
/*
   PQUEUE is the structure for the periodic data queues.
 */
typedef struct pqueue
 {
   PBUF*          current;
   PBUF*          reserve_list;
   char*          name;
   int            put_count;
   int            get_count;
   int            lock;
   int            lock_priority;
   int            processor;
   EVENT*         put_event;
  } PQUEUE;
```

Figure 3.7: Periodic Data Queue structure (PQUEUE).

by re-triggering itself each time it executes.

Scheduling events. Events can be scheduled to be triggered at specific times. The procedure `hic_schedule()` given an event and a time, adds the event to the scheduled event list which is maintained in ascending order of time. Periodically the timer interrupts (currently every millisecond) and readies every event on the list whose time has expired, these events are then removed from the list. Periodic events are automatically rescheduled by the dispatcher each time they are invoked.

3.6 Periodic Data Queues

Periodic data queues are discussed in the various examples of Section 2. They are similar in some respects to *sticky messages* used in the GEM operating system developed at Ohio State University [Schw85]. There are four procedures for accessing pdq's. To place data on a pdq a process calls `pq_reserve()` which returns a pointer to an empty buffer, it then places the data in the buffer, and then calls `pq_put()` to place the buffer on the pdq. `Pq_get()` returns a pointer to the most recent buffer put on the queue (by `pq_put()`). In general a buffer obtained by `pq_get()` is shared by multiple processes so modifying the buffer in any way is undesirable. When a process

is done with a buffer obtained by `pq_get()`, `pq_unget()` is invoked to release the buffer. Figure 3.7 contains the `PQUEUE` structure.

`Current` points to the current buffer, that is the buffer put by the most recent call to `pq_put()`. When a pdq is initialized `current` is null. But once a call to `pq_put()` has been made, `current` always points to the most recent buffer so that subsequent calls to `pq_get()` always succeed. This last statement is not totally true, there is an instance where the current buffer pointer will be null. This is discussed below.

`Reserve_list` points to a list of empty buffers. A call to `pq_reserve()` removes the first buffer from the list and returns it. When a buffer is freed by a call to `pq_unget()` it is placed on the reserve list if it is not the current buffer and there are no other processes using the buffer (i.e. there are no outstanding calls to `pq_get()` for this buffer).

`Name` points to an ASCII name for the pdq for debugging purposes.

`Put_count` and `get_count` are instrumentation counters which count the number of calls to `pq_put()` and `pq_get()` respectively.

`Lock` and `lock_priority` form the pdq locking mechanism. Each of the pdq procedures locks the pdq before manipulating the structure. Locking a pdq includes raising the processor interrupt priority on the processor preventing interrupts so no other process on the same processor can become active and access the pdq. `Lock_priority` is used to hold the previous processor priority while the pdq is locked. Locking also involves a *test-and-set* operation on the `lock` field so processes on other processors will wait (in a busy-wait loop) until the locking process unlocks the pdq.

`Processor` is the code for the processor containing the pdq. The current implementation of HIC requires that a pdq and its buffers reside on a single processor (i.e. in the local memory for that processor).

`Put_event` points to an event associated with the pdq. If `put_event` is non-null it is triggered each time `pq_put()` is invoked.

Figure 3.8 shows the structure of *periodic data buffers* or `PBUF`'s.

`Next` points to the next `PBUF` when this buffer is in the reserve list of a `pqueue`.

`Inuse` is a counter of the number of processes currently using this buffer. When this buffer is returned by `pq_get()` `inuse` is incremented and when it is released by `pq_unget()` `inuse` is decremented. If `inuse` is zero for the current buffer in a call to `pq_put()` then the current buffer is placed on the reserve list before the buffer passed is made the new current buffer. Also during `pq_unget()`, if the buffer being released is no longer the current buffer and its count is zero then it is placed on the reserve list.

```
/*
   PBUF is the structure for the Periodic Data Buffers.
 */
 typedef struct pbuf
 {
   struct pbuf*  next;
   int           inuse;
   int           event_id;
   int           procedure_id;
   int           time;
   int           processor;
   char*         data;
  }PBUF;
```

Figure 3.8: Periodic Data Buffer structure (PBUF).

`Event_id`, `procedure_id`, and `time` identify the time and source of a buffer. `Pq_put()` places the id's of the current event and procedure and the current time in the `PBUF` before it is made the current buffer. Thus a process getting a buffer can determine if the data is stale or where the data originated from.

`Processor` has the same meaning as in the `PQUEUE` structure.

`Data` points to the actual data in the buffer. Typically, a `PBUF` is assigned a data area when it is initialized and it never changes. However, it is possible to change the `data` pointer. It must, though, point to the same size data area.

A major consideration behind the design of periodic data queues is that they not cause processes to block. The four pdq access procedures never block (this is almost true, see below) however `pq_reserve()` and `pq_get()` can fail. In both cases if they fail they return a null pointer. Well designed programs will always check the returned pointer from these procedures. `Pq_reserve()` will fail if there is no buffer available. `Pq_get()` fails under two circumstances, one is after the pdq has been initialized and before any buffer has been made the current buffer, the other is a rather obscure situation which arises when the reserve list is empty and `pq_reserve()` is invoked. If `pq_reserve()` finds the reserve list empty it checks the current buffer and if it is not in use (`inuse` is zero) it is seized and given to the calling process. And thus there is no "current" buffer until `pq_put()` is invoked. If `pq_get()` is called during this interval it will return a null pointer.

There are two reasons `pq_reserve()` behaves this way. One is so that

a pdq can operate with only a single buffer allocated to it. Consider the trajectory pdq's from any of the examples in Section 2. In each case the trajectory pdq is accessed only by the planner and the output procedures and these are always executed sequentially, so a single buffer will suffice. If only one buffer is used and `pq_reserve()` could not seize the current buffer, then after the first cycle the planner would not be able to obtain an empty buffer from the trajectory pdq and therefore unable to place new values on the queue. This is because the first time through the planner would take the only buffer and make it current leaving the reserve list empty. Subsequent attempts to reserve a buffer would then fail. This is one reason `pq_reserve()` is allowed to seize the current buffer *if it is not in use.*

It's reasonable to ask: why use a pdq in this case, when only a single buffer is necessary? It is not necessary. The planner and output procedures could pass values via a simple shared data structure. Explicit synchronization is not even necessary since the procedures always execute in sequence. The reason for using the pdq is essentially a matter of standardization. In the examples, the planner and output procedures are distinct so that they can be individually replaced (see the discussion on the object servo in Section 2.2) and using a pdq to communicate between them provides a well-known standard interface. In addition, using a pdq means that an instrumentation or debugging process could sample the trajectories without disturbing the other two procedures (but more than one buffer would be required in this case). As further justification, it is not necessary that the planner and output procedures be part of the same sequence, they might be part of independent processes, they might even run on different processors.

The other reason `pq_reserve()` is allowed to seize the current buffer is to add a little robustness to the system since `pq_reserve()` is more likely to succeed in pathological cases such as an errant process holding on to too many buffers. Note that this does *not* guarantee that `pq_reserve()` will always succeed but merely improves the odds slightly.

It was stated above that pdq's do not cause events to block. This is not really true. The `lock` field in a `PQUEUE` is used to block access to the `PQUEUE` structure itself. It is a bit ironic after the lengthy discussion in Section 3.2 on blocking to introduce this form of blocking here. A pdq is a shared resource with the potential for simultaneous access so some protection and synchronization is required. In fact, this is the only place in HIC itself where a conflict arises. A *test-and-set* protocol allows a processor accessing a pdq to lock out access by processes on other processors. A locked out processor spins its wheels in a busy-wait loop which seems wasteful but the amount

of time a pdq is actually locked is very small. In particular, the time spent busy-waiting is much less than the amount of time it would take to suspend the process and enqueue it while waiting for the lock to be lifted. On the other hand though, a heavily used pdq can cause delays of unpredicatable length since there is no queue of waiting processes maintained and there is a race among waiting processes to seize a pdq when it is unlocked. Balancing the likelyhood of this pathological behavior against the overhead of a "fair" access system it was decided to take the low-overhead approach. When an event locks a pdq it disables all interrupts on its own processor. The reason for this is to prevent a higher level process on the same processor from attempting to access the same pdq which would lead to deadlock. This simple approach is costly since *all* higher level events on that processor are locked out, not just events that are in conflict. And since an event locks out interrupts before performing the test-and-set the lockout can spread from processor to processor. Still, in balance, the decision was to use the simple synchronization method.

To prevent failures in `pq_reserve()` and `pq_get()` the designer of a system must ensure that there are a sufficient number of buffers allocated to each pdq. The straight forward method is to examine each procedure that accesses a pdq and find the maximum number of buffers it may hold at one time, sum over all the procedures that access a pdq and allocate that number of buffers. In a properly drawn diagram such as in those in Section 2 this is the same as counting the number of arrowheads on lines into and from a pdq. It is also necessary to consider any instrumentation, monitoring, or debugging processes that might access pdq's, buffers must be allocated for these. Allocating this number of buffers ensures that there will be no failures (excluding trying to get a buffer before one has been put on a queue and assuming there are no bugs or system failures that cause buffers to be lost). In cases such as the input and planner routines of a servo loop it is clear that the two routines will not each have a buffer at the same time because they always run sequentially so it is not necessary to allocate a buffer for each procedure. For this reason, as mentioned above, the trajectory pdq's as shown need only one buffer allocated.

Merging Pdq's. Each periodic data queue can have an event associated with it that is triggered whenever a put operation is performed on the queue. This event is, of course, free to do anything but an example usage is to merge pdq's. Consider Figure 3.9 which shows two levels of control similar to the object and finger servos in Figure 2.3 on page 8. First note that the dia-

grams for the servo themselves have been simplified, the finger servo box now represents the finger servo sequence and the associated pdq's. The other servos are similarly simplified. To put the figure in perspective, consider the situation where the four-fingered Utah/MIT hand is grasping and manipulating an object with three of its fingers while the fourth finger is in free motion, moving perhaps to a point on the object where it will help establish a new grip. A strategy very similar to this was used by Maw-Kae Hor to rotate objects with a planar manipulator dubbed the Four Finger Manipulator [Hor87] (HIC was not used on these experiments). Now in the example, the object servo controls three fingers using the finger position and forces from the finger servo and producing target values in the grasp targets pdq. Independently, the free motion servo is controlling one finger again using the finger servo's actual values and producing free targets in the free target pdq. The merge event is associated with both the grasp pdq and the free pdq as their *put event* and is triggered each time data is put on either of the queues. It then takes the most recent data on each of its input queues and produces a single set of target values for the finger servo's target pdq.

This method adds overhead to the system since there are two extra calls to both `pq_put()` and `pq_get()` and the merge event must be invoked and data copied etc. But it has the great advantage that it can be largely transparent to the rest of the system. The buffers for the grasp targets and the free targets are identical to the finger servo buffers, thus the same object servo output can be "plugged" directly to the finger servo for tasks with no fingers in free motion and likewise if all the fingers are in free motion there is no need for the object servo and the merging, the free motion output can communicate directly with the finger servo. Since essentially all that is involved is the changing of pointers this sort of change can be accomplished on the fly. Referring back to the discussion of homogeneous manipulations in Section 2.2, this is precisely the sort of activity that might occur during the transitions from one homogeneous task to another.

Note that the object and free motion servos are truly independent. They need not have the same cycle time or be synchronized in any way. This is why the merge event is made the put event for both queues, so that updating either set of targets causes the finger servo's targets to be updated. What happens if the two servos produce their output at nearly the same time? Suppose, for instance, that the free motion output follows close behind the object servo output. Under the HIC scheduler, if they are so close in time that the merge event is triggered twice before it begins execution then the merge event will only execute once and will have the most up to date data

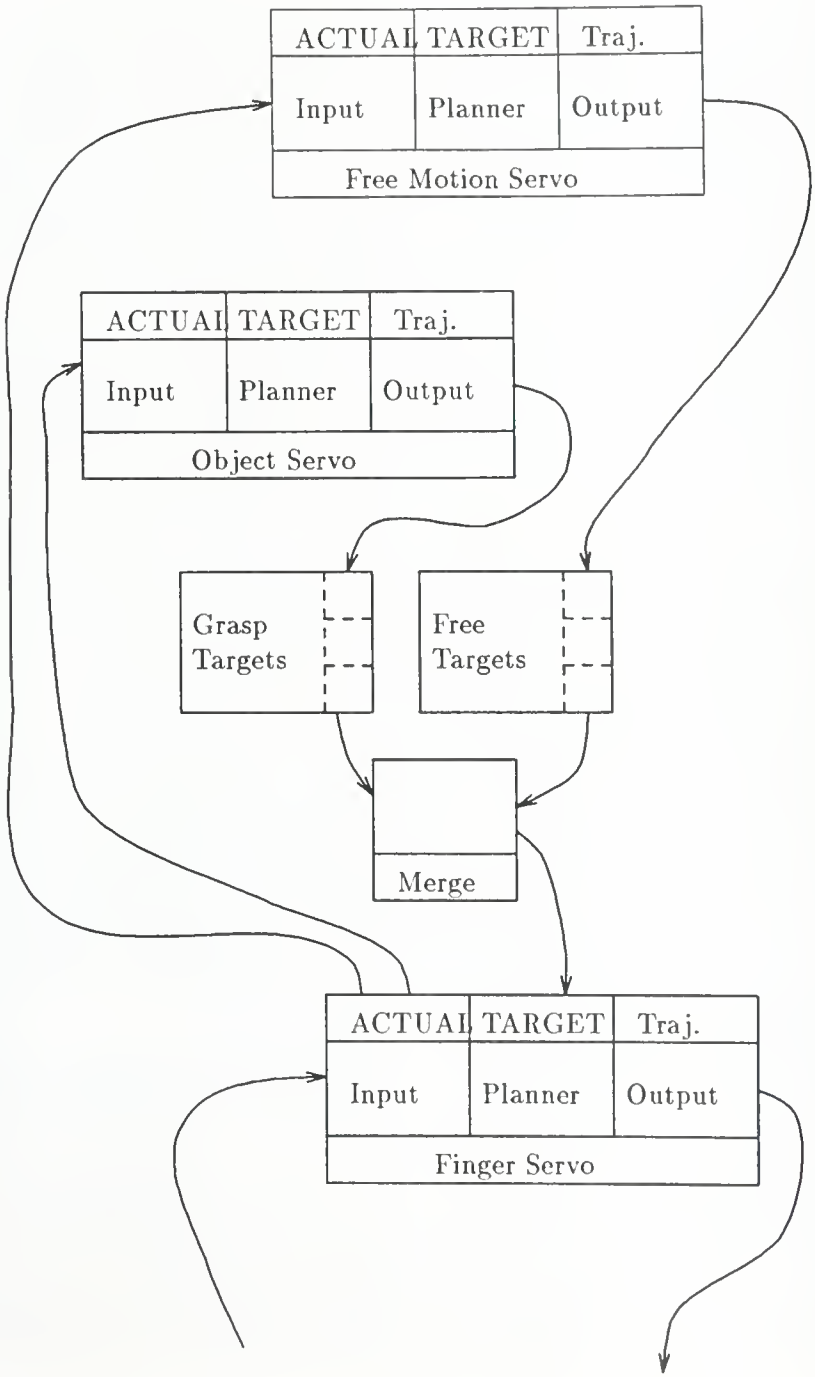


Figure 3.9: Merging Two Pdq's .

from each source. If the free motion output comes while the merge event is executing it will be performed a second time immediately upon completion. Thus two sets of targets would be produced in rapid succession for the finger servo the first will have the latest targets from the object servo and slightly out of date targets from the free motion servo, the second set of targets will have the latest targets from both. There is an assumption in this example that mixing slightly out of date targets with fresh targets is not a problem so that if the finger servo happens to pick up the mixed set of targets it will not cause the task to fail. If the finger in free motion is suitably far from the object or the system is suitably compliant this assumption is valid. If the task or system were sensitive to these mixed targets then some form of synchronization would be required between the object and free motion servos. If the two servos run at the same rate and in the same order each cycle, then a simple synchonization technique would be to associate the merge event with only one of the two input pdq's (the queue for the servo that finishes later each cycle). This way the merge event would only be invoked once each cycle and would have the most up-to-date data in each of its input queues.

3.7 Piggybacking HIC

HIC is not a general purpose operating system. It is suitable for a very specific part of real-time systems, namely the servo loops for low level control. At present, HIC is not an independent operating system, it runs on top of MIT's Condor which supplies the operating system nuts and bolts. HIC will always run as one part of a larger system.

Figure 3.1 shows the hardware configuration used to control the Utah/MIT hand. The Sun 3/160 controls the system. Programs are developed on the Sun, compiled, and downloaded to the Ironics processors which run the HIC/Condor combination. Current plans are that a single processor will be dedicated to the raw joint servo (see Section 2.1) running on HIC on top of Condor. Similarly, the finger and object servos will run on other processors (or perhaps on a single processor). Higher level supervisory tasks, are less likely to fit the HIC form and will require more sophisticated process management for file I/O and so on, though they will still have real-time constraints. One or more of the Ironics processors will be used for these tasks under the Sage operating system.

Sage [Salk88a] is a real-time operating system for supervisory control

developed at the NYU Robotics Laboratory. Compared to HIC, Sage is much more sophisticated. It provides true multi-tasking, memory management, inter-process communication and synchronization primitives, as well as support for file I/O while still providing real-time response where needed. It thus fills an important gap between the very low level functions of HIC and the general capabilities of the hosts development system. A scenario one might imagine is that task planning (i.e. deciding what is to be done) is done on the host processor. This involves complex decision making processes and intensive computation and to some extent is not bound by real-time constraints. The complex tasks will be managed by supervisory level programs running on the real-time processors (the Ironics) under Sage. These processes handle the sequencing and transition between the homogeneous tasks that make up the complex tasks. And finally, the homogeneous tasks are handled primarily by HIC tasks.

A variant of this architecture is to have HIC run as a single task under the control of another operating system such as Sage. Since HIC requires only a single stack it can run completely within a Sage process. This would be useful in a smaller system where a single processor is sufficient for both the low level servos and the supervisory control. The HIC task would run at a high priority to ensure the HIC tasks met their deadlines and the supervisory tasks would essentially run in the background.

In both of these scenarios the primary method of communication between HIC tasks and other tasks is via periodic data queues. Since pdq access never blocks it can be easily implemented within any operating system.

Process blocking is a factor here also. If HIC is running on top of Sage and one of the HIC events accesses a Sage facility *it may block*. In this case the entire HIC system will block since HIC appears to Sage as a single process. There is nothing inherently wrong with this but the programmer must be aware of the consequences.

Section 4

Run-time Library

4.1 Using HIC.

HIC consists of a set of C library routines and a header file. And since HIC runs on the Ironics processors a cross compiler is necessary to compile the programs. The HIC header file is `hic.h` which is in the directory `/usr/include/condor/remote`. When the ircc cross compiler is used the header file is included with the statement `#include <hic.h>`. The runtime library is `libhic.a` in `/usr/local/lib/condor/remote` and is accessed by the `-lhic` option on the compile statement. The following statement will compile the program `hicprog.c`:

```
% ircc -o hicprog -T 10000 hicprog.c -lhic
```

The `-T 10000` option causes the program to be located at address 0x10000 which is the normal starting address for programs on the Ironics. All of the options available on the Sun's C compiler are available on ircc.

After compilation the program can be downloaded to the Ironics with the command:

```
% dl68 -p 0 hicprog
```

where `-p 0` says to load to processor zero. To start a downloaded program use:

```
% istart 0 10000
```

where 0 is the processor and 10000 is the starting address.

For information on Condor see the Condor manual [Nara87].

4.2 HIC Procedures

This section and the next describe the important procedures in the HIC run-time library.

```
EVENT*
hic_sequence(event, name, id, interval, priority, flags, proc1, ..., procn, 0)
    EVENT*              event;
    char*               name;
    int                 id;
    TIME                interval;
    int                 priority;
    unsigned long       flags;
    PROC*               proc1;

                ⋮

    PROC*               procn;
```

creates an event or sequence and returns a pointer to the EVENT structure. If `event` is non-null it points to the EVENT structure to be used otherwise the structure is allocated by `hic_sequence()`. `Name` points to an ASCII name of the event for debugging purposes. `Id` is the identification number for the event, if it is zero or negative a unique id is automatically assigned. `Interval` is the interval in micro-seconds for the event, if it is periodic, zero otherwise. `Priority` is the nominal priority of the event. If it is negative or zero it is assigned based on the `interval`, note this is only meaningful for periodic events. See Section 3.3 for a discussion priority and rank. `Flags` are the initial settings of the `flags` field of the event's EVENTS structure. Two flags are typically set, EVENT_PERMITTED indicates that the event is allowed to run and EVENT_PERIODIC indicates that the event is periodic and thus it will be automatically re-scheduled each time it is triggered. If EVENT_PERIODIC is set the event must have a positive `interval`. `Proc1` through `procn` is the sequence of procedures (pointers to PROC structures) to be invoked when event is triggered. The list of procedures must be terminated by a null entry.

```
PROC*
hic_procedure(procedure, name, id, entry, argument, initial, reset)
    PROC*       procedure;
    char*       name;
    int         id;
```

```
    void        (*entry)();
    void*       argument;
    int         initial;
    int         reset;
```

creates a procedure and returns a pointer to the PROC structure for the procedure. Procedure points to the PROC structure to be used, if it is null then a structure is allocated by hic_procedure(). Name points to an ASCII name for the procedure. Id is the identifier for the procedure, if it is not positive a unique id is assigned. Entry points to the function to be invoked and argument is the argument to be passed to the function. Initial and reset are the values for the initial and reset fields of the procedure's PROC structure, see Section 3.4.

```
EVENT*
hic_prioritize_event(event_list, event)
    EVENT*      event_list;
    EVENT*      event;
```

inserts the event pointed to by event in the prioritized list event_list. A pointer to the top of the updated list is returned.

```
EVENT*
hic_events(event_list, event1, ..., eventn, 0)
    EVENT*      event_list;
    EVENT*      event1;

                :

   EVENT*       eventn;
```

inserts the events event1 through eventn in the prioritized list event_list. The list of events must be terminated with a null argument. A pointer to the top of the updated list is returned.

```
void
hic_initialize()
```

performs various initialization functions for the HIC system.

```
int
```

```
hic_schedule(event, time)
    EVENT*      event;
    TIME        time;
```

schedules the event indicated by `event` to be triggered at the absolute time, `time`.

```
void
hic_start(event_list)
    EVENT*      event_list;
```

starts the HIC system running. `Event_list` is the prioritized list of events (the first event has the highest priority). The sequence for invoking functions to start up HIC is as follows:

```
hic_initialize();                   /* Initialize. */
hic_procedure(...);                 /* Set up the procedures */
        :
hic_sequence(...);                  /* and events. */
        :
pq_create(...);                     /* Set up pdq's (see below). */
        :
hic_schedule(...);                  /* Schedule events. */
        :
event_list = hic_events (...);      /* Prioritize events. */
hic_start(event_list );             /* Start everything. */
```

The code following the call to `hic_start()` is the background task. It runs when there are no active HIC events. The background task must not exit.

```
void
hic_dispatch(event)
    EVENT*      event;
```

dispatches or triggers an event. This procedure is normally called from an interrupt handling procedure but can be called from user procedures if the *interrupts are disabled.* `Hic_dispatch()` sets the `EVENT_READY` flag in `event` and if the event has higher priority than the current event it is started.

4.3 Periodic Data Queues

These are the routines for accessing and manipulating periodic data queues.

```
PQUEUE*
pq_create(name, number_of_buffers, buffer_size, put_event)
    char*       name;
    int         number_of_buffers;
    int         buffer_size;
    EVENT*      put_event;
```

creates a periodic data queue. `Name` is the name of the queue in ASCII. It is included for debugging purposes. `Number_of_buffers` to allocate for the queue, see the discussion in Section 3.6 about this. `Buffer_size` is the size of each buffer in bytes. `Put_event` points to the put event (see Section 3.6) for the pdq.

```
PBUF*
pq_get(pq)
    PQUEUE*     pq;
```

gets the most recent buffer from the periodic data queue pointed to by `pq`. Assuming the returned buffer is called `pbuf` then `pbuf->data` points to the data in the buffer. Except in obscure cases (described in Section 3.6) `pq_get()` always succeeds and returns a pointer to the most recent buffer placed in `pq` by `pq_put()`. The user must treat this buffer as read-only since other processes may have hold pointers to the same buffer. The user may hold on to the pointer as long as necessary and when done `pq_unget()` is invoked to release the buffer.

```
void
pq_unget(pq, pbuf)
    PQUEUE*     pq;
    PBUF*       pbuf;
```

releases a periodic data buffer. `Pbuf` is a pointer obtained from `pq_get()` and `pq` points to the periodic data queue from which it was obtained. Each call to `pq_get()` must be balanced by a call to `pq_unget()` and the `pbuf` must be returned to the same pdq.

```
PBUF*
pq_reserve(pq)
    PQUEUE*     pq;
```

obtains a pointer to an empty buffer for the periodic data queue indicated by pq. The user is then should place data in the buffer in preparation for a call to pq_put().

```
void
pq_put(pq, pbuf)
    PQUEUE*     pq;
    PBUF*       pbuf;
```

places the buffer pointed to by pbuf on the pdq pq. Pbuf should have been obtained by a previous call to pq_reserve().

```
char*
pq_map_data(pbuf)
    PBUF*       pbuf;
```

returns a pointer to the data for the buffer, pbuf, mapped to the processor making the call. This is necessary for PBUF's residing on other processors. Frequently, the programmer knows that pbuf resides on the processor accessing it but it is good programming practice to use pq_map_data() anyway.

Section 5

Example Systems

This section contains examples of systems using HIC. Presented here are pieces of code accompanied by descriptions of the programs, the source code resides in the directory `/usr/src/local/hic`.

5.1 Raw Joint Servo

This is the low level joint servo as described in Section 2.1. It applies a combined force and position control scheme to the joints, in joint coordinates. The coordinates are simply those of the A/D and D/A converters in the system, that is 12-bit integers in the range -2048 through 2047. The major HIC structures used have been described in Sections 3.3 through 3.6 and illustrated in Figures 3.4 and 3.5 on pages 24 and 27.

To use the raw joint servo include the header file `<rj.h>` in the program and use the `librj.a` runtime library. For example, the limp program (source `limp3.c`)described in the next section is compiled with the following statement:

```
% ircc -o limp -T 10000 limp3.c -lrj -lhic
```

Figure 5.1 shows the structure holding global data related to the raw joint servo, `RJDATA`. The structure is for the most part pointers to periodic data queues that provide the interface to the joint servo, it is defined in the include file `rj.h`. The processor running the servo defines the structure and the pdq's, a process needing access to the raw joint data can request the address of the `RJDATA` structure via the `RJGET_RJDATA` mailbox request. Figure 5.2 shows how to obtain a pointer to the `RJDATA` structure. Where `RJPROCESSOR`

```
/*
   RJDATA is the data structure shared between the Ironics board running the raw joint
   servo and the other systems.
 */
typedef
struct rjdata
{
   int            servo_rate;
   EVENT*         servo_event;
   PQUEUE*        actual_pq;
   PQUEUE*        target_pq;
   PQUEUE*        parameters_pq;
   PQUEUE*        trajectory_pq;
 } RJDATA;
```

Figure 5.1: Joint Servo Global Data (RJDATA).

```
RJDATA*          rj;

          :
   rj = (RJDATA*) mbox_send_with_reply(RJPROCESSOR , RJGET_RJDATA, 0);
```

Figure 5.2: Obtaining Pointer to Raw Joint Data.

is the processor running the joint servo. See the Condor manual [Nara87] for a complete description of `mbox_send_with_reply()`. In this case it returns a pointer to the `RJDATA` structure. Note that the pointer is the address *local to* the raw joint processor and must be mapped into this processor's address space.

`Servo_rate` is the servo rate in cycles per second set by the servo initialization. Eventually it would be nice to be able to change the servo rate on the fly but at present there is no facility for this.

`Servo_event` points to the `EVENT` structure for the joint servo sequence.

`Actual_pq`, `target_pq`, `parameters_pq`, and `trajectory_pq` point to the four pdq's for the joint servo. They are used as in the three level control hierarchy shown in Figure 2.3. The structure of the pdq buffers is shown below.

```
/*
   RJACTUAL actual value periodic data buffer.
 */
 typedef
 struct rjactual
 {
   short position[NUMBER_OF_JOINTS];
   short extension[NUMBER_OF_JOINTS];
   short flexion[NUMBER_OF_JOINTS];
   short torque[NUMBER_OF_JOINTS];
  }RJACTUAL;


/*
   RJTARGET target value periodic data buffer.
 */
typedef
 struct rjtarget
 {
   short position[NUMBER_OF_JOINTS];
   short torque[NUMBER_OF_JOINTS];
  }RJTARGET;
```

Figure 5.3: Raw Joint Actual and Target Value Buffers.

Figures 5.3 and 5.4 show the data areas for the periodic data buffers. For all the values with the exception of `torque` and some of the parameters in `RJPARAMETERS` the units are simply the A/D and D/A values. `RJACTUAL` has `position` which is the angle of each joint, `extension` and `flexion` which are the measured strain on the two tendons for each joint, and `torque` which is computed from `extension` and `flexion`. `RJTARGET` has target position and torque (appropriately named) for each joint. The trajectory buffer, `RJTRAJECTORY`, has simply the commanded position for each joint.

The parameters, `RJPARAMETERS`, has six values for each joint. `Position_gain` and `torque_gain` are the gains applied to the errors in position and torque. There is a constant, `GAIN_DENOMINATOR` (currently 100), which is used to normalize the gains, thus the gains implicitly have a decimal point two positions to the left. `Delta_maximum` is maximum change in position allowed in any one cycle usually this is zero which means no limit. `Position_maximum` and `position_minimum` are the position limits for each

```
/*
   RJPARAMETERS parameter periodic data buffer.
 */
typedef
struct rjparameters
 {
   short position_gain[NUMBER_OF_JOINTS];
   short torque_gain[NUMBER_OF_JOINTS];
   short delta_maximum[NUMBER_OF_JOINTS];
   short position_maximum[NUMBER_OF_JOINTS];
   short position_minimum[NUMBER_OF_JOINTS];
   short threshold[NUMBER_OF_JOINTS];
  } RJPARAMETERS;



/*
   RJTRAJECTORY trajectory periodic data buffer.
 */
typedef
 struct rjtrajectory
 {
   short position[NUMBER_OF_JOINTS];
  } RJTRAJECTORY;
```

Figure 5.4: Raw Joint Parameters and Trajectory Buffers.

joint. `Threshold` is the minimum amount of change allowed in any cycle.

Initialization. The procedure `rjinitialize()`, in the raw joint library, initializes the joint servo. Among other things it sets up the servo event. Figure 5.5 shows the salient sections of this code. The data areas are allocated statically (this is not necessary since the procedures `hic_sequence()` and `hic_procedure()` could allocate them). The procedure `hic_sequence()` is called which sets up the event and returns a pointer to the event which is stored in the `servo_event` pointer in the `RJDATA` structure described above (note that `rj` points to the global `RJDATA` structure). The first argument is the address of the `EVENT` structure, the second is the name, and the third is the id number for the event. `Interval` is the interval in micro-seconds between instances of the event and `hic_standard_priority()` returns the standard priority for the event (depending on `interval`). The next ar-

```
/*
   Data areas for servo EVENT and PROC's.
 */
static EVENT    rjservo_event;
static PROC     rjinput_proc;
static PROC     rjplanner_proc;
static PROC     rjoutput_proc;

        ⋮

 /*
    Setup the servo event.
  */
  rj->servo_event =
      hic_sequence(&rjservo_event, "raw joint servo",
                   RJSERVO_ID, interval, hic_standard_priority(interval),
                   (EVENT_PERIODIC | EVENT_PERMITTED),
                   hic_procedure(&rjinput_proc, "raw joint input",
                                 RJINPUT_ID, rjinput, 0, 1, 1),
                   hic_procedure(&rjplanner_proc, "raw joint planner",
                                 RJPLANNER_ID, rjplanner, 0, 1, 1),
                   hic_procedure(&rjoutput_proc, "default rjoutput",
                                 RJOUTPUT_ID, rjoutput, 0, 1, 1),
                   0);
```

Figure 5.5: Servo Event Initialization.

gument is the initial flag settings. The event is a periodic event (EVENT_PERIODIC) and it is allowed to execute (EVENT_PERMITTED). The remaining arguments are pointers to the procedures making up the sequence terminated by a zero argument.

Each call to hic_procedure() returns a pointer to the PROC structure for the procedure. The arguments are a pointer to the structure, the name, the id, and a pointer to the C function to be invoked followed by the argument to be passed to the function. The last two arguments are values to put in the initial and reset fields of the PROC structure (see Section 3.4).

The initialization procedure also sets up the periodic data queues as shown in Figure 5.6. The four calls to pq_create() allocate space for the pdq's and their buffers and initialize them. In each case a pointer to the PQUEUE structure is returned and placed in the global RJDATA structure. For

```
/*
   Create the periodic data queues.
 */
rj->actual_pq = pq_create("rjactual queue", NUMBER_OF_RJACTUAL_BUFFERS,
                          sizeof(RJACTUAL), 0);
rj->target_pq = pq_create("rjtarget queue", NUMBER_OF_RJTARGET_BUFFERS,
                          sizeof(RJTARGET), 0);
rj->parameters_pq = pq_create("rjparameters queue",
                              NUMBER_OF_RJPARAMETERS_BUFFERS,
                              sizeof(RJPARAMETERS), 0);
rj->trajectory_pq = pq_create("rjtrajectory queue",
                              NUMBER_OF_RJTRAJECTORY_BUFFERS,
                              sizeof(RJTRAJECTORY), 0);

/*
   Initialize the parameters and targets.
 */
pbuf = pq_reserve(rj->parameters_pq);
*(RJPARAMETERS*) pq_map_data(pbuf) = default_parameters;
pq_put(rj->parameters_pq, pbuf);

pbuf = pq_reserve(rj->target_pq);
*(RJTARGET*) pq_map_data(pbuf) = initial_target;
pq_put(rj->target_pq, pbuf);
```

Figure 5.6: Joint Servo Pdq Initialization.

the servo to operate it needs some initial targets and control parameters. The next six statements take care of this. For both the parameters and targets a buffer is reserved (`pq_reserve()`), the initial values copied into the buffer and then these are put on the pdq (`pq_put()`).

Input. Figure 5.7 shows most of the servo input procedure, `rjinput()`. First an `RJACTUAL` buffer is reserved, and if that is successful then the raw data values are read from the A/D boards, three calls to `rjscan()` to get the positions, the extension tensions and the flexion tensions (not all shown in the figure). The torques on each joint are computed from the tensions (again this is abbreviated in the figure). Finally, the `RJACTUAL` buffer is placed back on the actual pdq.

Output. Figure 5.8 shows the servo's output procedure (Note that this figure and all remaining figures are at the end of the section) First a pointer

to the most recent trajectory data is obtained. If that is successful then we get a pointer to the position array, `position` and then send the values to the D/A converters in the loop. Finally, we release the `PBUF` with a call to `pq_unget()`.

Note that, in the general case the output procedure should be prepared for the possibility that the planner might fail to produce new trajectories in which case the output procedure would get the same set of trajectories twice. This is very unlikely given the structure of the joint servo sequence but it is a possibility in general. Because of the nature of the Utah/MIT hand's controller and because absolute positions are used as the trajectories there is no harm in applying the same trajectory twice, in this system. However, in some systems the trajectories may be incremental changes which if applied would be an error. There are two easy ways to guard against this sort of failure. One is simply to check the time stamp on the `PBUF` before using it and if it is the same as obtained in the previous cycle, don't apply the trajectories. The other is to put a flag or counter in the trajectory data structure that indicates that the buffer has been used. To modify the flag though requires use of `pq_reserve()` and `pq_put()` since it is a violation of the pdq protocol to modify a buffer obtained with `pq_get()`.

Planner. Figures 5.9 through 5.11 shows the planner procedure for the raw joint servo. The first of these figures shows the preparations for each cycle. The actual, target, and parameter buffers are obtained and if that is successful the output trajectory buffer is reserved. Then, if there are no problems we get pointers to the actual data arrays `position`, `torque`, `position_target`, etc. Figure 5.10 shows the actual planning loop. We step through the joints calculating for each one the adjustment in position (`delta`) based on the measured error. If `delta` is greater than the threshold for the joint then it is placed in the trajectory `PBUF` otherwise the current position is copied to the buffer. Then the pointers to the arrays are all incremented. At the completion of the loop `trajectory_pbuf` is put on the trajectory pdq. And finally, Figure 5.11 simply shows the cleanup at the end. All of the input buffers are released.

Again note the rather cavalier attitude towards errors. If there is any problem with obtaining input buffers of reserving an output buffer the planning cycle is simply aborted leaving the previous values on `trajectory_pq`. In the Utah/MIT hand controller this is acceptable because of the excellent design of the hand and its electronic controller. It is also possible because of the unlikelihood of a failure to obtain a buffer from one of the queues. In a

```
void
rjinput()
{

        :

 /*
    First reserve a PBUF from the actual queue. If we were successful continue, otherwise
    skip the whole thing. Point actual to the data area of pbuf. Rjscan() reads the
    A/D boards, first get the actual positions then the extension tensions and then the
    flexion tensions.

    Now for each joint, calculate the torque. We start with the outer joint (joint 3) of
    each finger and compute the torque as the strain times the radius of the pulley. For
    joint 2 the torque is the strain times radius plus the torque on joint 3 and likewise
    for joint 1. Joint 0 is independent of the other joints and is simply the strain times
    the radius.

    Finally, put the PBUF back on the queue, with the new values.
  */
  pbuf = pq_reserve(rj->actual_pq);
  if(pbuf)
    {
     actual = (RJACTUAL*) pq_map_data(pbuf);
     rjscan(DT1401_POSITION_BOARD(0), DT1401_POSITION_CHANNEL(0),
            DT1401_POSITION_CHANNEL(NUMBER_OF_JOINTS - 1),
            actual->position);

                    :

     extension = actual->extension + (NUMBER_OF_JOINTS - 1);
     flexion = actual->flexion + (NUMBER_OF_JOINTS - 1);
     torque = actual->torque + (NUMBER_OF_JOINTS - 1);

     accumulated_torque = RJ3_RADIUS * (*extension-- - *flexion--);
     *torque-- = accumulated_torque;
     accumulated_torque += RJ2_RADIUS * (*extension-- - *flexion--);
     *torque-- = accumulated_torque;
     accumulated_torque += RJ1_RADIUS * (*extension-- - *flexion--);
     *torque-- = accumulated_torque;
     *torque-- = RJ0_RADIUS * (*extension-- - *flexion--);

              :

    pq_put(rj->actual_pq, pbuf);
    }
}
```

Figure 5.7: Joint Servo Input Procedure.

system that is more sensitive to failure in the planner a more sophisticated recovery procedures would be required.

5.2 Limp

The limp program, shown in Figure 5.12, is the simplest complete program to use the raw joint servo. This program runs on one of the Ironics processors and simply invokes the joint servo. The default targets and parameters for the joint servo are such that the hand simply goes limp, that is the joints are totally compliant and react to any force by moving in the direction of the force.

The initialization in the program is straight forward, HIC is initialized, the servo is initialized, the servo event is scheduled, and the system is started. The code following `hic_start()` becomes the background task and the only requirement is that it *not* exit. Note that the pointer `rj` is defined in the header file `rj.h`.

Limp never changes the parameters or targets. However, a program on another processor can access the pdq's and thereby control the hand.

5.3 Rjedit

Rjedit is a Sun resident program for monitoring and modifying the raw joint servo's values, targets, and parameters. Most of the program deals with the display and user interface but it also shows how a program on the Sun can interact with HIC programs running on the Ironics processors. The procedure to run rjedit is to start the program on the Ironics (the limp program from the previous section for instance) and then run rjedit on the host.

Figure 5.13 show the initialization section concerned with the HIC programs. `Muse_init()` is the condor initialization procedure, it sets up the mailbox interrupt system on the Sun. The first argument specifies the processor number used to identify this process when communicating with the Ironics processors and the second argument specifies how the mailbox interrupts are to be handled. For more information about `muse_init()` see *The Condor Programmer's Manual – Version II* [Nara87].

Next `hic_initialize()` is called which takes care of HIC related initialization for the Sun process. `Rjdata()` returns a pointer to the `RJDATA` structure defined on the processor running the raw joint servo (`RJPROCESSOR`). It gets the pointer via a mailbox request and maps it into the Sun pro-

cess's address space. Rjedit is interested in three of the servo's pdq's, the actual, the target, and the parameter queues. The next three statements get the points to these pdq's. The pointers in the `RJDATA` structure pointed to by `rj` are pointers into the address space on `RJPROCESSOR`, the procedure `hic_map_from()` maps the pointers into the Sun process's address space.

Figure 5.14 shows snippets from the procedure `set_field_in_group()` which is used in rjedit to modify one of the servo's target values. First we obtain the needed `PBUF`'s, a copy of the current target values and a reserved buffer for the new targets. Note that the pointers have to be mapped into this process's address space, but that the procedures `pq_get()` and `pq_reserve()` take care of this. If we were successful, set the flag `got_pbuf` and then get pointers to the data areas for each of the `PBUF`'s. Here it is necessary to explicitly map the addresses. Now we copy the current targets into the new targets so that only the ones to be modified need be changed. At this point the copy of the current targets are no longer useful so we can release them (`pq_unget()`). Before actually modifying the targets we check the `got_pbuf` flag. Finally, if all is successful then the new targets are put on the target pdq with a call to `pq_put()`.

```
void
rjoutput()
{

        :

 /*
    Get the most recent PBUF from the trajectory queue. If we were successful fill it up.
  */
  pbuf = pq_get(rj->trajectory_pq);
  if(pbuf)
    {
     /*
        Get pointer to the data area of the trajectory PBUF. Get pointer to the first tra-
        jectory position. Get pointer to the first D/A converter (Note that we depend on
        the fact that the D/A converters are consecutive in memory). Put the trajectory
        positions in the D/A's.
      */
      trajectory = (RJTRAJECTORY*) pq_map_data(pbuf);
      position = trajectory->position;
      dac = dt1406->data;

      for(joint = 0; joint < NUMBER_OF_JOINTS; joint++)
        {
          *dac++ = (*position++ & DT1406_MASK);
        }

     /*
        Release the pbuf.
      */
      pq_unget(rj->trajectory_pq, pbuf);
    }
}
```

Figure 5.8: Joint Servo Output Procedure.

```
void
rjplanner()
{

        ⋮

 /*
    Get the input values.
  */
  actual_pbuf = pq_get(rj->actual_pq);
  target_pbuf = pq_get(rj->target_pq);
  parameters_pbuf = pq_get(rj->parameters_pq);

  if(actual_pbuf && target_pbuf && parameters_pbuf)
    {
     /*
        If we got the inputs reserve the output buffer.
      */
     trajectory_pbuf = pq_reserve(rj->trajectory_pq);
     if(trajectory_pbuf)
       {
        /*
           Get pointers to the actual data.
         */
         actual = (RJACTUAL*) pq_map_data(actual_pbuf);
         target = (RJTARGET*) pq_map_data(target_pbuf);
         parameters = (RJPARAMETERS*) pq_map_data(parameters_pbuf);
         trajectory = (RJTRAJECTORY*) pq_map_data(trajectory_pbuf);

         position = actual->position;
         torque = actual->torque;

         position_target = target->position;
         torque_target = target->torque;

         position_gain = parameters->position_gain;
         torque_gain = parameters->torque_gain;
         delta_maximum = parameters->delta_maximum;
         position_maximum = parameters->position_maximum;
         position_minimum = parameters->position_minimum;
         threshold = parameters->threshold;

         new_position = trajectory->position;
```

Figure 5.9: Joint Servo Planner Procedure – Setup.

```
        /*
           Compute the trajectory for each of the joints.
         */
         for(joint = 0; joint < NUMBER_OF_JOINTS; joint++)
           {
            /*
               Watch out! we have to convert to integers here to avoid overflow.
             */
             delta = (short)
                 (((((int) (*torque_target - *torque)) * (*torque_gain)) +
                   (((int) (*position_target - *position)) *
                    (*position_gain))) /
                  GAIN_DENOMINATOR);

             if(delta < 0)
                 {delta_sign = -1; delta = -delta;}
             else
                 delta_sign = 1;

             if((*delta_maximum > 0) && (delta > *delta_maximum))
                 delta = *delta_maximum;

             if(delta >= *threshold)
               {
                 *new_position = *position + (delta_sign * delta);
                 if(*new_position > *position_maximum)
                     *new_position = *position_maximum;
                 if(*new_position < *position_minimum)
                     *new_position = *position_minimum;
               }
             else
                 *new_position = *position;

             position++; torque++; position_target++; torque_target++;
             position_gain++; torque_gain++; delta_maximum++;
             position_maximum++; position_minimum++;
             threshold++; new_position++;
           }

         /*
            Put the PBUF back on the queue.
          */
          pq_put(rj->trajectory_pq, trajectory_pbuf);
        }
}
```

Figure 5.10: Joint Servo Planner Procedure – Planning.

```
  /*
     Release the input buffers.
   */
  if(actual_pbuf) pq_unget(rj->actual_pq, actual_pbuf);
  if(target_pbuf) pq_unget(rj->target_pq, target_pbuf);
  if(parameters_pbuf) pq_unget(rj->parameters_pq, parameters_pbuf);
}
```

Figure 5.11: Joint Servo Planner Procedure – Cleanup.

```
#include <stdio.h>
#include <hic.h>
#include <rj.h>
main()
{
  hic_initialize();
  rjinitialize();
  hic_schedule(rj->servo_event, (TIME) 0);
  hic_start(hic_prioritize_event(0, rj->servo_event));

  while(1)
    {
     /*
        Background task here.
      */
    }

  exit(0);
}
```

Figure 5.12: The Limp Program.

```
#include <stdio.h>
#include <condor/mbox.h>
#include <hic.h>
#include <hic_sun.h>
#include <rj.h>
#include <fedit.h>
#include "rjedit.h"

        :

RJDATA*         rj;
PQUEUE*          actual_pq;
PQUEUE*          target_pq;
PQUEUE*          parameters_pq;

        :

main()
 {
   muse_init(PROC_SUN_1, LISTEN_WITH_SIGNAL);
   hic_initialize();

   rj = rjdata();
   actual_pq = (PQUEUE*) hic_map_from(RJPROCESSOR, rj->actual_pq);
   target_pq = (PQUEUE*) hic_map_from(RJPROCESSOR, rj->target_pq);
   parameters_pq = (PQUEUE*) hic_map_from(RJPROCESSOR, rj->parameters_pq);

           :

 }
```

Figure 5.13: Rjedit – Initialization.

```
    pbuf = pq_get(target_pq);
    new_pbuf = pq_reserve(target_pq);

    if(pbuf && new_pbuf)
      {
       got_pbuf++;
       target = (RJTARGET*) hic_map_from(target_pq->processor,
                                         pbuf->data);
       new_target = (RJTARGET*) hic_map_from(target_pq->processor,
                                             new_pbuf->data);
       *new_target = *target;
      }
    if(pbuf) pq_unget(target_pq, pbuf);

                :
if(got_pbuf)
  {
   /*
      Modify the desired field in new_target.
    */

                :
  }
  if(new_pbuf) pq_put(target_pq, new_pbuf);
```

Figure 5.14: Rjedit – Pdq Access.

NYU COMPSCI TR-396
Clark, Dayton
Hierarchical control
system. c.1